Kea – A Dynamically Extensible and Configurable Operating System Kernel

Alistair C. Veitch and Norman C. Hutchinson

Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4
{aveitch,norm}@cs.ubc.ca

Abstract

Kea is a new operating system kernel which has been designed for maximum flexibility and performance in the areas of kernel and application extensibility and dynamic reconfiguration. Kea provides a means through which kernel services can be reconfigured, either on an application specific or system wide scale. We describe the design and implementation of these features, and report on some of our current research which relies on these abilities.

1 Introduction

Kea is a new operating system kernel being developed at the University of British Columbia which has been designed for maximum flexibility and performance in the areas of extensibility and dynamic reconfiguration, from both the kernel and application viewpoints. These capabilities are needed due to the specialized demands made on many modern systems and applications – requirements which may not have been foreseen by the system's designers. To illustrate the uses of dynamic reconfiguration and extensibility, consider the following diverse range of features which a modern operating system could be required to support: Mobile computing forces the operating system to deal with a dynamically changing environment and disconnected operation. If the system can dynamically reconfigure itself to match the new environment, it may be possible to realize substantial operational benefits. Multimedia systems (which could be either a single client application, or a dedicated server) require specialized network services, or guarantees, that the default protocol implementations may not provide. In these cases, giving the application the ability to configure in a new network protocol would be vital. Database systems, in order to achieve peak efficiency, often implement, or reimplement, services normally provided by the operating system, such as threads, memory management, and filesystems. By giving such systems the ability to reconfigure the kernel to include these services, even better performance should be achievable. Modern application technologies, such as Java [1], require sophisticated security in the form of access controls, that default operating system mechanisms may not provide. In this case, the ability to transparently interpose a new security service between the application and the operating system would be beneficial. Finally, the page replacement algorithms of many systems interact badly with the memory access patterns of applications, such as Lisp and some object-oriented systems that must do garbage collection or sequential reading of a mapped file. In these cases, the ability to install a per-application paging algorithm could also increase performance.

The Kea operating system has been designed to allow the development of sophisticated applications that require such specialized services, by providing a means through which applications can dynamically reconfigure, replace or extend existing services in an incremental and safe manner. The remainder of this paper gives a structural overview of the design and implementation of the mechanisms through which Kea achieves these goals and details some of the research we are currently conducting into dynamic reconfiguration, in which Kea is a vital component. In particular, section 2 introduces the Kea architecture, describing the features which make Kea reconfigurable. Section 3 gives a brief summary of the implementation and some performance figures. Section 4 then describes the experiments being conducted using Kea, and section 5 compares Kea to several other systems with similar goals. Finally, section 6 summarizes the paper.

2 Architecture

Kea is similar to many micro-kernel systems in that it provides lightweight abstractions of a machine's physical resources (memory, CPU, interrupts etc.), which can then be used to construct higher level services (such as network protocols). These higher level services communicate with applications and each other using some form of inter-process communication (IPC), often with some form of remote procedure call (RPC) layered on top. The inter-process communication mechanism is where Kea is primarily different from other systems, and is also where any reconfiguration takes place. Kea's form of RPC is completely general and by allowing the destination of an RPC to be changed independently of the caller, also enables

services to be dynamically and transparently reconfigured or extended, providing both the operating system and application designer a great degree of flexibility. Several base Kea abstractions, *domains*, *threads*, *inter-domain calls* and *portals*, combine to make this reconfiguration possible.

2.1 Domains

Domains are the simplest and most standard of Kea's abstractions. Domains are simply virtual address spaces, offering mappings from virtual to physical memory. Physical pages may be shared, copied, or mapped between domains, and can have different protection attributes associated with them. In this respect, Kea is no different from most modern kernels. Domains also serve as the key protection point in the system, in the sense that domains are the entities to which system resources are allocated.

2.2 Threads

A thread is the context in which code execution takes place, and from the applications point of view provides similar services and properties to those in other operating systems. Internally however, Kea separates the kernel thread state, such as scheduler and security information, from the current execution state (domain, machine registers and stack). The execution state is called an activation, and each thread can have multiple activations associated with it, organized as a stack, of which only the activation at the top of the stack is eligible to execute. This organization is very similar to that of Spring [2] and the "migrating threads" developed for Mach 4.0 [3] (from which, in the interests of reducing confusion and the growth of new jargon, the activation term is borrowed).

2.3 IDC

The reasoning behind the dual nature of threads is to facilitate the service structuring mechanism, which is achieved through the use of an RPC-like abstraction. Instead of building a conventional IPC/RPC layer, with its associated inefficiencies, Kea provides a mechanism which has all the semantics of a procedure call, and which is valid between domains, called an *Inter-Domain Call*, or IDC. When an IDC is performed, a new activation (in the destination domain) is created for the current thread, and pushed onto the thread's activation stack. In addition, any parameters associated with the call are also available in the destination domain, either on the stack or in registers, depending on the calling convention of the machine. In this fashion, the same thread continues execution, albeit in a different domain.

While the effect of an IDC is very similar to a traditional RPC (although IDC's are limited to a single machine[1]) the means through which it is achieved is quite different, and IDC is intrinsically more efficient as it eliminates several unnecessary steps often made in a general RPC system. Firstly, IDC is not based on a lower IPC layer, and so removes one level of processing from the call path. Secondly, it is not necessary to use a canonical data encoding, since all parameters have the same encoding on both caller and callee. These benefits come from not having to assume the callee might be on a different machine. While other systems, notably LRPC [5], have also made similar optimizations, IDC's are further optimized by the thread model, in that the scheduler does not have to be involved in the IDC – control passes directly between domains, but the same thread remains executing. Internally, the system can also keep caches of activations and their associated stacks in order to further decrease the time taken for an IDC.

2.4 Portals

Portals are the Kea abstraction used to represent an IDC entry point. They provide a handle into both a domain and an entry point within that domain, and once created (typically, but not always, by the domain containing the code to be executed for the IDC), they can be freely passed around between domains. By using the system call `portal_invoke()`[2], the first argument to which is a portal identifier, an IDC is performed.

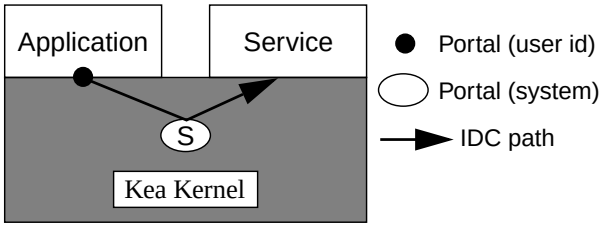

Figure 1. Portal Invocation

Figure 1 illustrates the effect of a portal invocation. An application calls `portal_invoke()`, using a previously acquired portal identifier (small dark circle), and the system performs the appropriate IDC to the service implementing that IDC. The underlying implementation maps the local (application level) portal identifier into a system portal structure (labelled oval), which determines the number and size of arguments actually copied to the destination service. The service could reside in any domain – including the kernel or the calling application.

1. To extend Kea for use in a distributed environment, it is envisioned that a standard proxy mechanism [4] can probably be used.
2. It is interesting to note that `portal_invoke()` is the only system call in Kea. All other base services create portals for their entry points at system initialization.

Semantically, portal invocation is almost identical to a procedure call, and is sufficient to layer services. To give Kea more structure however, we have designed services as sets of *interfaces*. An interface is a definition of the allowable interactions between the implementor of a service and the clients of that service. A service is defined as a set of procedure definitions, from which it is possible to generate simple client stubs that use portal invocations. Clients are written to conform to the interface, and do not need to know about portal invocation. By making interfaces a high-level abstraction, we are able to "hide" the underlying portal mechanism, and unify all the entry points for a service into a single entity. By making each service relatively fine-grained, and providing a means through which each service's interface can be specified, hierarchies of services can be built. For example, low level disk drivers present only a block read/write interface, specific filesystem interfaces are built on this, and the global filesystem/name-space on top of this.

2.5 Portal Remapping

The most important property of portals, and the one that makes reconfiguration possible, is that they can be dynamically remapped to a different domain/entry-point at run-time. This remapping is transparent to the user of the portal, i.e. the application uses the same portal identifier, but any IDC through that portal will result in the activation of a different service (that should perform the same function as the original, albeit improved in some fashion). A simple example is shown in Figure 2. Here, an application has been using a particular file service (F), but the system administrator has made a new file service (F'), with the same semantics, but improved performance properties, available. Either the application itself, or the system administrator, can cause the application portals[1] referring to the original file server to be remapped to the new ones. Internally, the bindings between the original portal identifiers are modified to refer to the equivalent system portals in the new file service. IDC's through the original portals now execute code in the new service. This type of remapping could also be used to test new software, or for simple upgrade reasons (i.e. to fix bugs). The important features are that the applications portal identifiers do not change, and that the remapping is transparent to the application, and does not affect other applications using the original service. It is this ability to perform dynamic remapping that is the primary difference between Kea's IDC mechanism, and other similar RPC mechanisms, such as doors in Spring [2].

1. For clarity, the figures only show a single portal remapping.

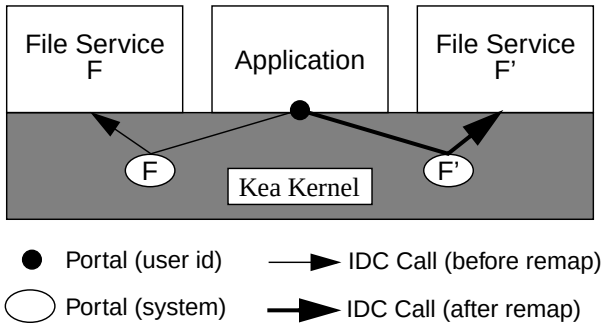

Figure 2. Simple Portal Remapping

Imagine that the service depicted in Figure 1 is a file service, but that the application wishes to have the data in its files compressed. In a normal system the application itself would have to explicitly call a compression routine on every write request, and the corresponding decompression function on every read. Using Kea however, a compression service can be defined, with the same semantics as the file service (i.e. supplying the same functions), but that transparently compresses all data passing through it. The application could then remap the file system portals to refer to this compression service, while the compression service itself continues to use the original file service. The configuration of such a system is shown in Figure 3, where the application portal originally referred to the portal structure labelled F, but has been remapped to that offered by the compression service (C). This style of remapping operation (which is identical to that shown in Figure 2, with the addition of an extra invocation by the replaced domain), can be used to support any form of stackable service or interposition agents [6].

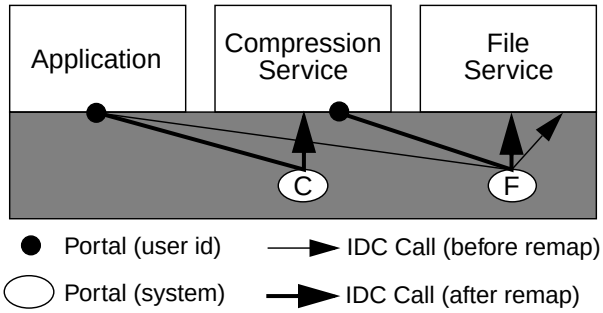

Figure 3. Stacked Portal Remapping

The final, and most complex, remapping supported by Kea is shown in Figure 4. Here, two applications are using the virtual memory services, through some arbitrary chain of portal invocations. This is shown implicitly in the diagram by the stacking of the applications upon the virtual

memory service. The virtual memory service uses, in its turn, a policy module which selects pages to be paged out when necessary. If application B can determine its own memory usage patterns, it is possible that significant savings in CPU and disk traffic could be realized by letting it install its own policy module. Kea allows this to happen, by letting B define its own page selection service, and remapping the portal used by the VM service to call the page selection service, in such a way that portal invocations using that portal, that are on behalf of domain B, use the new page selection service, while all other IDC's using that portal, such as those made by A, continue to use the default service. This is a form of indirect mapping, in that the portal remapped has no association with the original application domain. Other uses of this type of remapping could be used for process scheduling, buffer cache management, or any other part of the system where a policy decision must be made, but the caller may be separated by several portal invocations from the original domain. As this remapping type is dependent upon the domain in which a thread originated, we refer to it as a *domain-rooted* remapping.

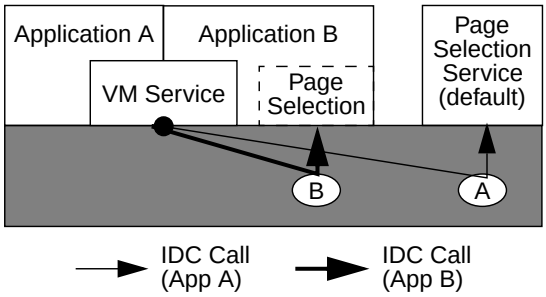

Figure 4. Domain-rooted remapping

In addition to the per-domain remappings described above, it is equally possible to remap portals on a system-wide basis, i.e. to transparently remap portals for all users of a service. Given these remapping possibilities, the true configurability of a Kea-based system becomes obvious – any service in the system can be transparently replaced at any time with an equivalent service. In addition to the examples given above, this also opens such possibilities as continuous systems operation (software upgrades by service replacement) and operating systems emulation (by installing a new service). Several diverse application areas could benefit from user-level reconfiguration, and many examples of the uses to which such flexibility could be put have been cited. These include security enhancements [7], scheduling support in parallel systems [8] and application level virtual memory management [9,10,11].

3 Implementation and Performance

Kea has been under development since June of 1994, and currently exists as a bootable kernel for Intel 80x86 processors in IBM-compatible P.C.s. Development has also been performed on sun3 and sun4 machines, although the versions extant on these machine tend to lag behind those on the Intel platform. All the base Kea abstractions are implemented, and experimentation has begun on dynamic reconfiguration of user-level services.

While the existing version is a research prototype only, it must still be at least comparable in performance to similar systems, in order that reasonable comparisons on the utility of the reconfiguration techniques proposed can be made. The most important performance-related component of Kea is IDC times (although it can be argued that such measurements are becoming increasingly irrelevant [12]), so we have made various measurements of the time taken for IDC's of various data sizes, and compared these to similar operations for the Mach and L3 microkernels [13]. The results from this are shown in Table 1. All the times shown were measured on a 486-DX50 PC compatible machine, and are in microseconds. Mach and L3 times involved an RPC with a fixed size data block parameter (transferred from caller to callee), while Kea times were for the equivalent IDC with the same size parameter. As can be seen, Kea is certainly faster than Mach, but not nearly as fast as L3. There are several reasons for the latter. Firstly, L3 is machine-specific – the kernel is written entirely in assembler, reducing the overhead considerably, compared to the portable code used in Mach and Kea. Additionally, the L3 measurements are made with a minimal system, with the system operating almost entirely out of L1 cached memory, which may not be valid for a system running several applications. Secondly, IDC is a far more general mechanism than IPC, in which there are hidden overheads, due to the fact that the receiver of a message has to unmarshall that message into the appropriate data structures. Also, with client/server IPC based systems, servers often have a single receiver that does a switch on the incoming message, and calls the appropriate function dependent on the message type. Thirdly, we anticipate that we will be able to substantially tune various Kea subsystems, in particular the virtual memory system and argument copying mechanisms, leading to a corresponding performance increase for IDC – the context switch time is currently the biggest component (about 70%) of the IDC time. This is further attested to by IDC's that only go into the kernel (which are exactly analogous to system calls). A "thread_id()" system call takes approximately 11µs, comparing reasonably well to the getpid() time of 8µs for Linux on the same machine. These results indicate that while IDC times may not be as fast as possible, they should

be adequate for evaluating Kea's potential, and will not substantially interfere with any further research into the use of such systems for dynamic reconfiguration.

As mentioned previously, the application level portal identifiers are mapped into a kernel level portal structure, which describes the data that needs to be copied for that particular portal. There are two stages to this mapping. The first is to perform a hashed lookup on the threads original domain, in order to determine if a domain-rooted remapping has been made, and if so, to return the remapped portal structure. If no such remapping has been made, the mapping is accomplished using a simple array lookup. These operations are relatively inexpensive (only 2µs of the measured IDC time) and remapping a portal involves only changing the stored system portal identifier within the relevant data structure. As a consequence, there are no performance implications from any portal remapping operation.

Table 1. IDC times (microseconds)

	Data Size (kilobytes)		
	0K	1K	4K
Mach	230	320	560
Kea	87	156	320
L3	10	32	175

4 Future Directions

While we have implemented the Kea system design as described above, we have yet to perform any substantial experiments on dynamic reconfiguration. We are currently investigating several different areas, notably dynamic relocation of services between user and kernel space, dynamic network protocol configuration, and filesystem layering. Each of these offers practical and relevant demonstrations as to the utility of operating system reconfiguration, and as such is deserving of further study.

4.1 Service Relocation

Kea, like many other microkernel based systems, suffers from efficiency problems caused by the communications mechanism. No matter how fast an IDC performs, it will still be many orders of magnitude slower than the equivalent procedure call occurring within a single address space. This is because of the overheads associated with data copying and switching of address spaces, which results in TLB and cache flushes, and a concomitant decrease in performance [14]. However, services running in separate address spaces do have distinct advantages. Firstly, the service can be debugged using a standard debugger, important during system development. Secondly, running the service in its own domain provides greater security to the system as a whole – any faulty behaviour by the service is confined in its effects to the local domain, and cannot crash the system as a whole.

However, these advantages are primarily important to the system developer only. Once services such as filesystems have been debugged, there is little need for separation into disjoint domains, and more need for optimal performance, certainly when it is considered that performance is often the sole criteria by which operating systems are judged. In addition, services such as filesystems are usually of a highly trusted nature anyway, and the extra security risks incurred by placing such services in the kernel are minimal. Given these circumstances, it would be useful if services could be dynamically loaded into, and out of, the kernel. This would allow dynamic replacement of services that required upgrading (reducing computer downtime) as well as facilitating the development and debugging of these services – certainly being able to disregard the eventual run-time location of services (kernel *vs.* user space) eases development considerably. This decoupling of modularity, protection and run-time environment is an important feature in Kea's design, although it is not entirely unique to Kea [15].

At present, we are investigating the possibilities for dynamic kernel service reconfiguration. We currently have device drivers that use the same source code, regardless of whether they run at kernel or user level. We currently need to recompile before these drivers can be shifted into or out of the kernel, since not all the low-level interfaces are currently specified as such (i.e. they rely on kernel/user library code, rather than portal-remappable versions), and we have yet to target the compiler to produce relocatable code (which is necessary, since remapped binaries may be loaded and run at any memory address)[1]. Once these problems have been addressed, it will be possible to transparently migrate such code between the kernel and user levels, in order to gain the advantages of both.

4.2 Network Reconfiguration

The *x*-kernel is a configurable kernel designed to support experimentation in network protocols and distributed programming [17]. The modular structure of the *x*-kernel and its protocols make it the perfect vehicle for experimenting with dynamic loading and reconfiguration of network protocols. We have completed a port of the *x*-kernel to Kea, and currently use it as the base networking system. The *x*-kernel's internal structure matches well with

1. We are however investigating the possibility of using current research in heterogeneous process migration [16] to remove this requirement.

the service primitives provided by Kea, in particular the "thread per message" paradigm fits well with IDCs and the *x*-kernel APIs map cleanly into Kea interfaces. We intend to modify the *x*-kernel to enable the dynamic loading of new networking protocols into a running system, and to perform any necessary reconfiguration of protocol layering. We also expect to be able to use it to provide further performance comparisons for equivalent services running in kernel and user domains.

4.3 Filesystem layering

The final area in which we wish to experiment is filesystems. In most modern operating systems, filesystems are layered on top of a generic layer (e.g. vnodes [18]), which is in turn layered on device drivers. By making these interfaces explicit within Kea, we hope to be able to reconfigure any of these layers, but intend to concentrate on the upper filesystem, and provide examples of interface remapping for supporting such things as transparently compressed, replicated and distributed filesystems.

In addition, Kea is currently being used in the development of a multimedia file server [19]. Kea provides unique advantages to such demanding applications, allowing them efficient access to kernel resources, or even allowing dedicated systems to be incorporated into the kernel, where they can achieve maximum efficiency. In particular, the multimedia server requires fast, low-level access to disks, dedicated network bandwidth, and customized scheduling, none of which are generally available in a standard operating system such as UNIX.

5 Related Work

Extensibility and configurability have been goals of operating systems for some time, but it is only recently that researchers have turned to the production of systems with these as primary goals, rather than secondary features. For example, Multics [20] provided a means (*gates*) through which applications could specify which procedure segments it would use. However, the binding between segments was statically performed at load-time, with no means by which an application could modify a segment. The operating system itself was also constructed at a high level, and did not provide access to, let alone manipulation of, the lower level features of the system.

The Synthesis operating system [21] provides enhanced application performance through run-time generation and optimization of code which interfaces to operating system services. An example is the file system, where an *open* call returns not a handle to a file, but a section of code optimized for performing operations on that particular file, for that particular client. While this method provides enhanced performance for applications, it does not provide the application with a means of modifying the default interfaces themselves.

The GNU Hurd project [22] is an attempt to structure a module-based system on top of the Mach 3.0 kernel. Like Kea, modules will be dynamically replaceable by arbitrary applications. Unlike Kea however, modules are heavyweight structures (e.g. whole file systems), which violates the principle of incrementality, and so reduces the general applicability of the module replacement facility. It is unlikely that any system built on top of Mach, or similar microkernels, can provide true application specific features without substantial modification, as the microkernel itself provides no means for application-specific mappings of fine grain modules, particularly those that may reside within services composed of an aggregation of cooperating modules.

SPIN, a new microkernel design with application specificity as its primary goal, is currently under development at the University of Washington [23]. SPIN allows applications to install low-level system services into the kernel, so that they can receive notification of, or take action on, certain kernel events. These events may be as disparate as a page fault, the reception of a network packet destined for the application, or context switches affecting the application. Extensions to the kernel are written in Modula-3, and they rely on the safety features of the language and compiler to verify, as far as possible, the code for integrity. Higher level flexibility, that is application specific functionality that does not have to be present in the kernel, is accomplished through the use of application specific libraries. These approaches provide reconfiguration for applications requiring low-level services but the interfaces to, and provisions for, higher level flexibility or global system reconfiguration are more restrictive than Kea's. The work being done for SPIN's features is however valuable, and they may prove to be important components of future reconfigurable systems.

Finally, some projects [24,25] are investigating the use of software fault isolation [26] techniques for the safe linking of code directly into the kernel. These projects also show promise, and we may adopt some of their methods for co-locating code, particularly in the cases where we wish to provide some enhanced security.

6 Conclusions

This paper describes the design and implementation of Kea: an extensible, reconfigurable operating system kernel. Kea incorporates several new features, notably portals (supporting IDC's) and portal remapping, whereby the system can be easily reconfigured. We have shown that IDCs are comparable in efficiency to other IPC-based systems, implying that Kea will also be comparable in

overall performance, and should certainly be sufficient for investigation into dynamic system reconfiguration.

In addition, we intend to do further research into the reconfigurable aspects of Kea's design, extending our understanding of what system components can (or should) be reconfigured, and the most useful ways in which this can be done. We will concentrate on three areas: user/kernel service remapping, dynamic network protocol remapping, and filesystem layering.

In summary, we feel that modern systems and applications require system reconfigurability in order to achieve maximum performance. We believe that the Kea architecture, in particular inter-domain calls and portal remapping, is ideal for the support of such reconfiguration of systems at both the kernel and application layers.

7 References

[1] J. Gosling and H. McGilton, *The Java Language Environment*. http://www.javasoft.com/whitePaper/java-whitepaper-1.html

[2] G. Hamilton and P. Kougiouris, The Spring Nucleus: A Microkernel for Objects. *Proceedings of the 1993 Summer USENIX Conference*, June 1993, pp. 147-159

[3] B. Ford and J. Lepreau, Evolving Mach 3.0 to a Migrating Thread Model. *Proceedings of the 1994 Winter USENIX Conference*, January 1994, pp. 97-114

[4] M. Shapiro, Structure and Encapsulation in Distributed Systems: The Proxy Principle. *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986, pp. 198-204

[5] B.N. Bershad, T.E. Anderson, E.D. Lazowska and H.M. Levy, Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1), February 1990, pp. 37-55

[6] M.B. Jones, Interposition Agents: Transparently Interposing User Code at the System Interface. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 80-93

[7] B.N. Bershad & C.B. Pinkerton, Watchdogs – Extending the UNIX File System. *Computing Systems*, 1(2), Spring 1988, pp. 169-188

[8] T.E. Anderson, B.N. Bershad, E.D. Lazowska and H.M. Levy, Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1), February 1992, pp. 53-79

[9] K.Krueger, D. Loftesness, A. Vahdat and T. Anderson, Tools for the Development of Application-Specific Virtual Memory Management. *Proceedings of the 1993 Conference on Object Oriented Programming Systems, Languages and Architectures*, 1993, pp. 48-64

[10] A.W. Appel & K. Li, Virtual Memory Primitives for User Programs. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, April 1991, pp. 96-107

[11] S. Sechrest and Y. Park, User Level Physical Memory Management for Mach. *Proceedings of the USENIX Mach Symposium*, November 1991, pp. 189-199

[12] B.N. Bershad. The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems. *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, April 1992, pp. 205-212

[13] J. Liedtke, Improving IPC by Kernel Design. *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993, pp. 175-188

[14] J.C. Mogul and A. Borg, The Effect of Context Switches on Cache Performance. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, April 1991, pp. 75-84

[15] P. Druschel, L.L. Peterson and N.C. Hutchinson, Modularity and Protection Should be Decoupled. *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992, pp. 95-97

[16] P. Smith and N.C. Hutchinson, *Heterogeneous Process Migration: The Tui System*. Submitted for publication.

[17] N.C. Hutchinson and L.L. Peterson, Design of the *x*-Kernel. *Proceedings of the 1988 SIGCOMM*, August 1988, pp. 65-75

[18] S.R. Kleiman, Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Proceedings of the 1986 USENIX Conference*, pp. 238-247

[19] G. Neufeld, D. Makaroff and N. Hutchinson, The Design of a Variable Bit-Rate Continuous Media Server. *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Durham, NH, April 1995, pp. 375-378

[20] E.I. Organick, *The Multics System: An Examination of It's Structure*. The MIT Press, Cambridge, MA, 1972

[21] H. Massalin and C. Pu, Threads and Input/Output in the Synthesis Kernel. *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989, pp. 191-201

[22] M. Bushnell, Towards a New Strategy of OS Design. In *The January 1994 GNU's Bulletin*.

[23] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers and S. Eggers, Extensibility, Safety and Performance in the SPIN Operating System., *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995, pp. 267-284

[24] D. Engler, M.F. Kaashoek and K. O'Toole, The Operating System Kernel as a Secure Programmable Machine. Proceedings of the *Fifteenth Symposium on Operating Systems Principles*, December 1995, pp. 251-266

[25] C. Small and M. Seltzer, *VINO: An Integrated Platform for Operating System and Database Research*. Technical Report TR-30-94, Harvard University, 1994.

[26] R. Wahbe, S. Lucco, T.E. Anderson and S.L. Graham, Efficient Fault Isolation. *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, December 1995, pp. 203-216