K42: Building a Complete Operating System

Orran Krieger † Marc Auslander † Bryan Rosenburg † Robert W. Wisniewski †
Jimi Xenidis † Dilma Da Silva † Michal Ostrowski † Jonathan Appavoo †
Maria Butrico † Mark Mergen † Amos Waterland † Volkmar Uhlig †

ABSTRACT

K42 is one of the few recent research projects that is examining operating system design structure issues in the context of new whole-system design. K42 is open source and was designed from the ground up to perform well and to be scalable, customizable, and maintainable. The project was begun in 1996 by a team at IBM Research. Over the last nine years there has been a development effort on K42 from between six to twenty researchers and developers across IBM, collaborating universities, and national laboratories. K42 supports the Linux API and ABI, and is able to run unmodified Linux applications and libraries. The approach we took in K42 to achieve scalability and customizability has been successful.

The project has produced positive research results, has resulted in contributions to Linux and the Xen hypervisor on Power, and continues to be a rich platform for exploring system software technology. Today, K42, is one of the key exploratory platforms in the DOE's FAST-OS program, is being used as a prototyping vehicle in IBM's PERCS project, and is being used by universities and national labs for exploratory research . In this paper, we provide insight into building an entire system by discussing the motivation and history of K42, describing its fundamental technologies, and presenting an overview of the research directions we have been pursuing.

Categories and Subject Descriptors

D.4.0 [**Operating Systems**]: General; D.4.1 [**Operating Systems**]: Process Managment; D.4.1 [**Operating Systems**]: Process Managment—*Multiprocessing*; D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**Operating Systems**]: File Systems Management; D.4.4 [**Operating Systems**]: Communications Management; D.4.7 [**Operating Systems**]: Oragnization and Design; D.4.8 [**Operating Systems**]: Performance

General Terms

Algorithms,Performance,Design

†IBM T. J. Watson Research Center
This work was supported in part by a DARPA PERCS grant contract number NBCH30390004

EuroSys'06, April 18-21, 2006, Leuven, Belgium.

Keywords

operating system design,scalable operating systems, customizable operating systems

1. BACKGROUND

In 1996 we began K42 to explore a new operating system design structure for scalability, customizability, and maintainability in the context of large-scope or whole-system research issues. K42's design was based on current software and hardware technology and on predictions of where those technologies were headed. In this section we describe those predictions and discuss the resulting technology decisions. At the end of the paper, we review how the predictions changed over the life of the project and the resulting changes in the technical directions of the project.

1.1 Technology predictions

Key predictions we made in 1996 were:

1. *Microsoft Windows would dominate the client space, and would increasingly dominate server systems.* By the mid 1990s, predictions made by leading consulting firms indicated Unix would disappear from all but high-end servers and Windows would dominate most markets.
2. *Multiprocessors would become increasingly important at both the high and low end.* For the high end, projects in academia [24, 1, 20] demonstrated that large scale NUMA multiprocessors are feasible and can be developed to be price/performance competitive with distributed systems. For the low end, the increasing number of transistors were yielding smaller improvements to single cores, and it seemed that the ever increasing density of transistors would instead be used for more cores and threads on a chip.
3. *The cost of maintaining and enhancing current operating systems would grow increasingly prohibitive over time.* Existing operating systems were designed as monolithic systems, with data structures and policy implementations spread across the system. Such global knowledge makes maintenance difficult because it is hard to ensure that the effect of a small change is contained. The difficulty of innovating in such systems was inhibiting OS research and making it very difficult to affect commercial systems. The places where existing operating systems did have well defined object-oriented interfaces, e.g., vnode layer, streams, drivers, etc., allowed greater innovation.
4. *Customizability and extensibility would become increasingly critical to operating systems.* There was a considerable amount of research in the 1990s on customizable

and extensible operating systems. Projects like Exokernel[14], Spin[7], and Vino[31] demonstrated that performance gains could be achieved if the OS is customized to the needs of specific applications. In the context of large multiprocessors with potentially many simultaneous jobs, customizability seemed even more critical.

5. *Within five years all machines would be 64 bit.* MIPS and Alpha had already demonstrated that the incremental cost of supporting 64-bit addressing is minimal on chip area and power dissipation. Because 64-bit addressing is beneficial for some software, it seemed that 32-bit platforms would soon disappear. Announcements by chip manufacturers supported this. One piece of software that can benefit from a 64-bit architecture is the operating system. Assuming 64-bit addressing allows optimizations throughout the operating system, e.g., mapping all physical memory, adding state and lock bits to pointers, avoiding complicated data structures by using potentially large virtual ranges rather than hashing, etc.

We now discuss the technology directions these predictions led us to pursue.

1.2 Technology directions

Many operating system research initiatives chose to explore new ideas through incremental changes to existing systems. We believed that the changes in technology since the existing operating systems were designed were significant enough, and that the cost of maintaining those systems was high enough, that operating system research from a clean slate should be undertaken.

We designed a system that was similar in many ways to the Exokernel, where much functionality is implemented in libraries in the applications' own address spaces. This structure was based on the premise that the more function that was moved out of the kernel, the easier it would be to provide specialized libraries to adapt to applications' needs. More importantly, we felt that micro-kernel systems like Workplace OS [23] failed in part because of the complexity needed in each server to support multiple personalities. It seemed that mapping from a specific OS personality to base OS services in the application address space could simplify the task and reduce the overhead.

One of the key design decisions we made in the system was to use object-oriented (OO) design pervasively. Work from the University of Toronto's Tornado project [17] had demonstrated that an OO methodology that encourages local implementations was important for achieving good multiprocessor performance. Stated alternatively, global data structures, policies, and implementations do not scale well; local implementations are needed. While Tornado focused primarily on exploiting object orientation to improve multiprocessor performance, the K42 project used it also for its software engineering advantages of reducing maintenance costs and enabling innovation.

Fault containment and recovery is a significant issue for operating systems that run across large numbers of processors. The cellular approach adopted by Hive[11] and Hurricane[37] had been deemed by both groups of researchers to have too high a performance cost. In K42 we decided to explore a model we called *Application Managers*. On large systems, multiple OSes of varying sizes would be multiplexed. An application that did not require many resources would run on an OS instance scaled to its needs. Our approach differed from Disco [19] in that fault containment boundaries were between OSes that were time multiplexed rather than space multiplexed. We assumed that the multiplexing would be managed by cooperative minimal kernels, not imposed by an external entity.

2. INTRODUCTION AND GOALS

The K42 project is an open-source research operating system kernel incorporating innovative mechanisms, novel policies, and modern programming technologies.

In K42 we are examining the issues of designing a complete operating system kernel. It is one of the few recent research projects[25, 16, 9, 22] that addresses large-scope operating system research problems. While not every line of code in K42 was written from scratch, K42 was designed from the ground up to address scalability and dynamic customization of system services. K42 supports the Linux API and ABI [2], allowing it to run unmodified Linux applications and libraries.

The principles that guide our design are 1) structuring the system using modular, object-oriented code, 2) designing the system to scale to very large shared-memory multiprocessors, 3) leveraging the performance advantages of 64-bit processors, 4) avoiding centralized code paths and global locks and data structures, 5) moving system functionality to application libraries that reside in the applications' address spaces, and 6) moving system functionality from the kernel to server processes. While we believe in these design principles, our primary emphasis is on performance. Therefore, as needed, we have made compromises rather than carrying these design philosophies to extremes in order to fully explore their ramifications.

Goals of the K42 project include:

Performance: 1) Scale up to run well on large multiprocessors and support large-scale applications efficiently. 2) Scale down to run well on small multiprocessors. 3) Support small-scale applications as efficiently on large multiprocessors as on small multiprocessors.

Customizability: 1) Allow applications to determine (by choosing from existing components or by writing new ones) how the operating system manages their resources. 2) Autonomically have the system adapt to changing workload characteristics.

Applicability 1) Effectively support a wide variety of systems and problem domains. 2) Make it easy to modify the operating system to support new processor and system architectures. 3) Support systems ranging from embedded processors to high-end enterprise servers.

Wide availability: 1) Be available to a large open-source and research community. 2) Make it easy to add specialized components for experimenting with policies and implementation strategies. 3) Open up for experimentation parts of the system that are traditionally accessible only to experts.

Having described our overarching goals and principles, in the rest of the paper we present a summary of our different research directions we pursued in K42. We begin in Section 3 with a discussion of K42's scalability. Then, in Section 4, we describe memory allocation, which is key to

achieving good scalability. In Section 5 we discuss how the object-oriented model allowed us to achieve a dynamically customizable system. Clustered objects, described in Section 6, are important to achieving good scalable performance and providing an infrastructure for customization. Section 7 describes K42's user-level scheduler, its threading model and efficient inter-process communication infrastructure. File system research directions are described in Section 8. A key to achieving good performance is understanding performance. We describe K42's performance monitoring infrastructure in Section 9. To explore complete operating system issues, the ability to be able to run applications and have a user base is critical. We addressed this requirement by providing Linux compatibility as described in Section 10. In Section 11 we describe the use of this environment by discussing application use, debugging, and K42's development model. In K42, we considered *Application Managers* (now commonly called VMMs Virtual Machines Managers or hypervisors) as a way to achieve greater availability. In the last several years this work has taken on even wider scope. We describe the virtualization work associated with K42 in Section 12. We present concluding remarks in Section 13.

3. SCALABILITY

An early emphasis in the project was achieving good performance on scalable hardware, and scalability remains a cornerstone of K42. K42 is designed and structured to scale as hardware resources are increased. The techniques used in K42 provide a runtime structure that can be efficiently mapped to the underlying multiprocessor structure, enabling scalable performance.

Achieving scalable performance requires minimizing sharing, thereby maximizing locality. On shared memory multiprocessors (SMPs), locality refers to the degree to which locks are contended and data is shared between threads running on different processors. The less contention on locks and the less data sharing, the higher the locality. Maximizing locality on SMPs is critical, because even minute amounts of (possibly false) sharing can have a profound negative impact on performance and scalability [17, 10].

We use four primary techniques for structuring our software to promote and enable scalability:

PPC: A client-server model is used in K42 to represent and service the demand on the system. A system service is placed within an appropriate protection domain (address space) and clients make requests to the service via a Protected Procedure Call (PPC). Like the Tornado PPC [17], a call from a client to a server that, like a procedure call, crosses from one address space to another and back with the following properties: 1) client requests are always serviced on their local processor, 2) clients and servers share the processor in a manner similar to handoff scheduling, and 3) there are as many threads of control in the server as there are outstanding client requests.

Locality-aware dynamic memory allocation: Dynamic memory allocation is extensively used in the construction of K42 services to ensure that memory resources grow with demand both in size and location. The K42 memory allocators employ per-processor pools to ensure that memory allocations are localized to the processors servicing a request for which the memory was allocated.

Object decomposition: K42 uses an object-based software architecture to implement its services. The model encourages developers to implement services that can scale as demand increases. A service is structured as a set of dynamic interconnected object instances that are lazily constructed to represent the resources that a unique set of requests require. For example, mapping a portion of a process's address space to a file would result in the construction of several objects unique to that process, file, and mapping. Thus, page faults of the process to that address range would result in execution of methods only on the objects constructed for that process, file, and mapping. Other mappings would result in other independent objects being constructed. The object architecture acts as a natural way of leveraging the locality-aware dynamic memory allocation. Objects are created on demand on the processor on which the request was made. Thus, they consume memory local to those processors and are accessed with good locality.

Clustered objects: K42 uses an SMP object model that encourages and enables the construction of scalable software services. Each object can be optimized with respect to the concurrent access patterns expected. For example, a multi-threaded application, such as a web server or a parallel scientific application, causes concurrent accesses to the Process object instance representing that application. This occurs, for example, when its threads on different processors page fault. The Clustered Object model provides a standard way for implementing in a scalable manner the concurrently-accessed objects using distribution, replication, and partitioning of data structures and algorithms.

4. MEMORY MANAGEMENT

The goals of high performance, multiprocessor scalability, and customizability, have impacted the design of memory management in K42. In addition to helping fulfill the overall goals, additional goals such as a pageable kernel helped define the design as well. The design goals and constraints include:

Locality: Avoid global data structures and locks. Manage each processor separately.

Uniform buffer cache: Have a single mechanism for caching file and computational data in page frames, partitioned for locality.

Enable user-mode scheduling: Reflect page faults and page-fault completions to the faulting process. Do not block for page I/O in the kernel.

Bounded kernel resources: Queue delayed work and resume it when indicated by interrupt or IPC, rather than blocking threads in the kernel.

External file servers: Provide interfaces so that processes other than the kernel can manage file system resources.

Pageable kernel: Provide for paging much of the kernel, especially the paging-system data needed to represent non-kernel processes.

Unix semantics: Provide implementations of fork and copy-on-write mappings.

Customizability: Allow customized paging implementations for specific applications and allow different behavior for different regions within a given address space.

Processor-specific memory: Provide a space of virtual memory locations that are backed by different page frames on different processors.

NUMA and multiple page size support: Keep physical memory close to where it is used. Use large pages to reduce hardware and software costs of virtual memory.

To illustrate how an object-oriented model is used in a kernel, we describe the objects that are used to implement memory management in K42, what their functions are, and how they interact. Each K42 process contains a single address space. The address space is made up of Regions, each of which spans a range of contiguous virtual addresses in the process's address space. A Region maps its virtual range onto a range of offsets in a "file". We use the term file even for computational storage, although a special version of the "file" implements this case. A process also contains the hardware-specific information needed to map virtual addresses to page frames. We represent this memory structure using K42 Clustered Objects connected by K42 object references.

The Process object is the root of the object tree for each process in the kernel. It contains a list of Region objects and refers to the hardware-specific information.

Each Region object represents the mapping from a contiguous range of virtual addresses in the process address space to a contiguous range of offsets in a file.

The File Representative is the kernel realization of a file, used to call the external file system to do I/O, and used by client processes, for example, to memory map the file.

The File Cache Manager (FCM) controls the page frames assigned to contain file content in memory. It implements the local paging policy for the file and supports Region requests to make file offsets addressable in virtual memory.

The Page Manager (PM) controls the allocation of page frames to FCMs, and thus implements the global aspects of paging policy for all FCMs.

The Hardware Address Translator (HAT) manages the hardware representation of an address space. The SegmentHAT manages the representation of a hardware segment. Segments are of hardware-dependent size and alignment, for example 256 megabytes on PowerPC. The hardware data they manage might be a segment's page table or a PowerPC virtual segment id. SegmentHATs can be private or shared among multiple address spaces.

This particular decomposition was chosen because it was judged to separate mechanisms and policy that could be independently customized and composed. For example, a normal Region, or one specialized to implement processor-specific memory by including the processor number in its mapping of virtual address to file offsets, can each be connected to different kinds of files.

5. CUSTOMIZATION

K42 was designed so that each instance of a resource is managed by a different set of one or more object instances. For example, each open file in K42 has a different set of instantiated file object instances that provide the bits for that file. This is true for most other resources such as memory regions, processes, pipes, etc. Using different object instances for different resources allows the way in which the operating system manages resources on behalf of an application to be customized. Specifically, different applications that are running simultaneously can have different resource management policies applied to the same type of resource, such as memory, file, etc. Further, even within an application, different memory regions, or different files, etc., can be managed by different objects providing different customized policies.

This infrastructure also allows K42 to vary the level at which the hardware and software layer interfaces are implemented. For example, on different hardware platforms the interface can move up or down depending on the available hardware features. This is also true for software boundaries, e.g., the interface to glibc. Applications that are interested in specialized services can call customized interfaces of the K42 objects implementing the particular policy that the application needs rather than through the glibc layer.

Dynamic customization

The ability to customize how the operating system manages an application's resources can be used to improve application performance. The capability to modify the resource management on the fly provides even greater ability to match the needs of the application, but also provides a qualitatively different capability of improving the availability, upgradability, and security of the system.

There are a variety of reasons for modifying a deployed operating system. A common example is component upgrades, particularly patches that fix security holes. Other examples include dynamic monitoring, system specializations, adaptive performance enhancements, and integration of third-party modules. Usually, when they are supported, distinct case-by-case mechanisms are used for each example.

K42's dynamic customization support is divided into *hot swapping* [34] and *dynamic upgrade* [4, 5, 6]. *Hot swapping replaces an active component instance with a new component instance that provides the same interface and functionality. Internal state from the old component is transfered to the new one, and external references are relinked. Thus, hot swapping allows component replacement without disrupting the rest of the system and without placing additional requirements on the clients of the component.* Dynamic upgrade uses hot swapping to replace all of the objects in the system providing a given service. If an upgrade was made to the object representing the process, for example, the dynamic upgrade would be used to replace the objects in the system representing each process. There can be a potentially large number of objects to swap and thus it is important to be able to perform the swap lazily.

Applying dynamic customization

Hot swapping and dynamic upgrade are general tools that provide a foundation for dynamic OS improvement. The remainder of this section discusses several categories of common OS enhancements that are supported by hot swapping and dynamic update and how those OS enhancements may be implemented with hot swapping or dynamic update.

Patches and updates: As security holes, bugs, and performance anomalies are identified and fixed, deployed systems must be repaired. With dynamic upgrade, a patch can be applied to a system immediately without the need for down-time. This capability avoids having to make a trade-off among availability and correctness, security, and better

performance.

Adaptive algorithms: For many OS resources, different algorithms have better or worse performance under different conditions. Adaptive algorithms are designed to combine the best attributes of different algorithms by monitoring when a particular algorithm would be best and using the preferred algorithm at the correct time. Using dynamic customization, developers can create adaptive algorithms in a modular fashion, using several separate components. Although in some cases implementing such an adaptive algorithm may be simple, this approach allows adaptive algorithms to be updated and expanded while the system is running. Each independent algorithm can be developed as a separate component and hot swapped in when appropriate.

Specializing the common case: For many individual algorithms, the common code path is simple and can be implemented efficiently. However, supporting all of the complex, uncommon cases often makes the implementation difficult. To handle these cases, a system with dynamic customization can hot swap between a component specialized for the common case and the standard component that handles all cases. Another way of getting this behavior is with an if statement at the top of a component with both implementations. A hot-swapping approach separates the two implementations, simplifying testing by reducing internal states and increasing performance by reducing negative cache effects of the uncommon case code.

Dynamic monitoring: Instrumentation gives developers and administrators useful information about system anomalies but introduces overheads that are unnecessary during normal operation. To reduce this overhead, systems provide "dynamic" monitoring using knobs to turn instrumentation on and off. Hot swapping allows monitoring and profiling instrumentation to be added when and where it is needed, and removed when unnecessary. In addition to reducing overhead in normal operation, hot swapping removes the need for developers to a priori know where probes would be useful. Further, many probes are generic (e.g., timing each function call, counting the number of parallel requests to a component). Such probes can be implemented once, avoiding code replication across components.

Application-specific optimizations: Specializations for an application are a well-known way of improving a particular application's performance based on knowledge only held by the application [13, 15, 18, 38]. Using dynamic customization, an application can provide a new specialized component and swap it with the existing component implementation. This allows applications to optimize any component in the system without requiring system developers to add explicit hooks.

Third-party modules and Linux adoption: An increasingly common form of dynamic customization is loadable kernel modules. It is common to download modules from the web to provide functionality for specialized hardware components. As value-add kernel modules are produced, such as "hardened" security modules [21, 30], the Linux module interface may evolve from its initial focus on supporting device drivers toward providing a general API for hot swapping of code in Linux. The mechanisms described in this paper are a natural endpoint of this evolution, and the transition has begun; we have worked with Linux developers to implement a kernel module removal scheme using quiescence [27].

Summary

Dynamic customization is a useful tool that provides benefits to developers, administrators, applications, and the system itself. Each individual example can be implemented in other ways. However, a generic infrastructure for hot swapping can support all of them with a single mechanism. By integrating this infrastructure into the core of an OS, the OS becomes considerably more amenable to subsequent change.

6. CLUSTERED OBJECTS

Clustered Objects [3] are a crucial component to achieving scalability in K42, and they form the base infrastructure on which customizability support is built. The term "cluster" is derived from the fact that an object resides on one or more processors and services requests for one, some, or all of the processors. As we have described earlier, object orientation is a fundamental aspect of K42. Clustered objects build on standard object-oriented features such as inheritance, polymorphism, and specialization. Clustered Objects also provide multi-threading and semi-automatic garbage collection.

In addition to standard object-oriented features, Cluster Objects provide an interface that facilitates the design and implementation of scalable services. In particular, the implementation is hidden behind a user-visible interface in such a manner that the underlying service may be implemented on one, a subset, or all the processors as appropriate for that particular service. To illustrate this we describe a simple example of a counter object implementation.

Clustered Objects allow each object instance to be decomposed into per-processor Representatives (or Reps). A Representative can be associated with a cluster of processors of arbitrary size; they are not necessarily one per processor. A simple example would be a counter object, that has one data member, `val`, and methods `inc`, `dec`, and `getVal`. Externally, a single instance of the counter is visible, but internally, the implementation of the counter could be distributed across a number of Representatives, each local to a processor. An invocation of a method of the Clustered Object's interface `inc` or `dec` on a processor is automatically and transparently directed to the Representative local to the invoking processor. If common operations are `inc` and `dec` then the most efficient implementation would be for each Rep to maintain their `val` and only when `getVal` is invoked do the Reps need to communicate. This avoids unnecessary sharing and ensures good `inc` performance. If, however, the common operation is `getVal` then a shared implementation of `val` will perform better. This decision is transparent to the clients because their view is the interface exported by the clustered object.

To provide additional insight into how Clustered Objects help achieve scalability we describe the core aspects of their implementation. A more thorough treatment is available [3]. Each Clustered Object is identified by a unique interface to which every implementation conforms. An implementation of a Clustered Object consists of a Representative definition and a *Root* definition. The Representative definition of a Clustered Object defines the (potentially) per-processor portion of the Clustered Object. The Root class defines the global portions of an instance of the Clustered Object. Every instance of a Clustered Object has exactly one instance of its Root class that serves as the internal central anchor or "root" of the instance. The methods of a Rep can access the shared data and methods of the Clustered Object via its

Root pointer.

At run-time, an instance of a given Clustered Object is created by instantiating the desired Root class. Instantiating the Root establishes a unique Clustered Object Identifier (COID) that is used by clients to access the newly created instance. To the client code, a COID appears to be a pointer to an instance of the Rep Class. To provide code isolation this fact is hidden from the client code. There is a single shared table of Root pointers called the Global Translation Table and a set of Rep pointer tables called Local Translation Tables, one per processor. The Global Translation Table is split into per-processor regions, while the Local Translation Table resides at the same address on each processor by the use of processor-specific memory. This allows requests to Reps to be transparently directed to the local instance. Local Reps are created lazily when needed by a given cluster of processors.

The map of processors to Reps is controlled by the Root Object. A shared implementation can be achieved with a Root that maintains one Rep and uses it for every processor that accesses the Clustered Object instance. Distributed implementations can be realized with a Root that allocates a new Rep for some number (or cluster) of processors, and complete distribution is achieved by a Root that allocates a new Rep for every accessing processor. There are standard K42 Root classes which handle these scenarios.

7. USER-LEVEL IMPLEMENTATION OF KERNEL FUNCTIONALITY

As mentioned in the introduction, one of the research directions in K42 has been moving functionality traditionally implemented in the kernel into application space. This can be used, for example, to avoid system calls for `poll()` and `select()`. It can also be used to reduce the amount of storage and pinned memory required by the kernel. One piece of kernel functionality we have moved to in user-space in K42 is that of thread-scheduling. The kernel dispatches address spaces while a user-level scheduler dispatches threads. With threading models implemented at user-level, some aspect of communication can also be moved out of the kernel.

7.1 Scheduling

The scheduler is partitioned between the kernel and application-level libraries. The K42 kernel schedules entities called dispatchers, and dispatchers schedule threads. A process consists of an address space and one or more dispatchers. Within an address space, all threads that should be indistinguishable as far as kernel scheduling is concerned are handled by one dispatcher.

A process might use multiple dispatchers to either attain real parallelism on a multiprocessor, or to establish differing scheduling characteristics, such as priorities or qualities-of-service, for different sets of threads.

A process does not need multiple dispatchers simply to cover page-fault, I/O, or system-service latencies, or to provide threads for programming convenience. These requirements can be met using multiple user-level threads running on a single dispatcher. The dispatcher abstraction allows individual threads to block for page faults or system services without the dispatcher losing control of the processor. When a thread suffers an out-of-core page fault (a fault that cannot be resolved by simply mapping or copying a page already in memory, or zero-filling a new page), its dispatcher receives an upcall that allows it to suspend the faulting thread and run something else. A subsequent completion notification tells the dispatcher that the faulting thread can be resumed. System services provided by the kernel and other servers are invoked via protected procedure calls (PPCs). A thread making a PPC is blocked until the PPC returns, but its dispatcher remains runnable, allowing it to regain control and run other threads, should the PPC get delayed in the server for any reason. With this design, dispatchers tie up kernel resources, e.g., pinned memory; threads do not. A process that creates thousands of threads for programming convenience has no more impact on the kernel than a single-threaded process.

The kernel scheduler operates independently on each processor and determines which dispatcher is to run on the processor at any given time. It supports a small number of fixed priority classes and uses a weighted proportional-share algorithm to schedule dispatchers within a priority class. Any runnable thread of a higher priority class will always supersede any thread of a lower priority class. The small number of fixed priority classes are used to provide differing service guarantees to mixed workloads consisting of time-sharing, soft-real-time, gang-scheduled, and background applications. For example, priority classes are used to ensure that soft-real-time applications get the resources they have requested no matter how many time-sharing applications happen to be running, while the remaining processing resources are distributed among the time-sharing programs on a proportional-share basis. Also, for example, any task placed in the background priority class will really be in the background and will not take cpu time from any time-shared task. Though the choice of scheduling algorithm within each class is flexible, we use proportional share to provide understandable scheduling guarantees.

The user-level scheduler implements a threading model on top of the dispatcher abstraction provided by the kernel. The model allows different applications to have different threading models. The thread library is responsible for choosing a thread to run when the kernel gives control of the processor to a dispatcher. It is responsible for suspending and resuming a thread when it suffers a page fault, starting new threads to handle incoming messages, and handling asynchronous events such as timeouts. The thread library provides services for creating new threads as well as the basic suspend and resume operations that are needed for the implementation of locks and other higher-level synchronization facilities. Standard threading interfaces such as Posix Threads can be supported efficiently on top of the default threading library.

7.2 Interprocess and intraprocess communication

K42 provides both synchronous and asynchronous messaging primitives for communicating between processes, and it provides a fast signaling mechanism to facilitate communication between dispatchers within a process.

The primary interprocess communication mechanism in K42 is the protected procedure call, or PPC, which is the invocation of a method on an object that resides in another address space. The mechanism is designed to enable scalability (as discussed in Section 3) and allow efficient communication between entities in different address spaces but on the same physical processor. Calls are generally made from a client process to a server process that the client trusts to perform

some service. The client thread making the call is blocked until the call returns. Control is transferred to a dispatcher in the server process, and a new thread is created to execute the specified method. The thread terminates when the method returns, and control is returned to the client process.

Objects that a server wants to export are specified with an annotated C++ class declaration, which is processed by an interface generator. The interface generator produces a stub object on which the client can invoke methods and an interface object that invokes the corresponding method on the real object in the server. The stub and interface objects marshal and demarshal, parameters and return values, and use kernel primitives to accomplish the call and return of control. The processor registers are used to transfer data from caller to callee and back, and for this reason the dispatchers involved in a PPC are expected to be running on the same processor. For moderate amounts of data, i.e., for data too large to fit in the registers but smaller than a page, K42 provides a logical extension to the register set which we call the PPC page. Each processor has a unique physical page that is read-write mapped at a well-known virtual address into every process that runs on the processor. The contents of the page are treated as registers (i.e., saved and restored on context switch), although a process can dynamically specify how much of the page it's actually using to cut down on the save/restore costs. On a PPC, the contents of the PPC page are preserved from caller to callee and back, as are the real registers, so the page can be used for parameter and return value passing.

K42 also provides an asynchronous interprocess communication mechanism that can be used by server processes to signal untrusted clients. Like a PPC, an asynchronous call looks like a remote method invocation, however for asynchronous IPC, a message describing the call is injected into the target process and control remains with the sender. The sending thread is not blocked, and no reply message is generated or expected. Asynchronous calls are limited to a small number of parameter bytes and are generally used to inform a client that new data is available or that some other state change has occurred. The target dispatcher need not be on the same processor as the sender.

K42 provides a fast signaling mechanism to facilitate communication between the dispatchers belonging to a single process. Such dispatchers share an address space so data transport is not a problem. Message queues or other structures can be implemented in shared memory but a signaling mechanism is needed so that dispatchers do not have to waste time polling for incoming messages. In K42, a thread in one dispatcher can request a soft interrupt in another dispatcher in the same process. The soft interrupt will give control to the target dispatcher, allowing it to process the signal and switch to a more important thread if it was enabled by the signal.

8. FILE SYSTEMS

K42's file-system-independent layer performs name space caching on a per-file-system basis with fine-grain locking. Part of path name resolution is performed in user-level libraries that cache mount point information.

K42's file system (KFS) [33, 12] KFS runs on Linux and embodies many K42 philosophies including object-oriented design, customization within and between processes, moving kernel functionality into server processes, avoiding large-grain locking, and not using global data structures. For example, for KFS running on K42 there is no global page cache or buffer cache; instead, for each file, there is an independent object that caches the blocks of that file. When running on Linux, KFS is integrated with Linux's page and buffer cache.

We designed KFS to allow for fine-grained adaptability of file services, with customizability at the granularity of files and directories in order to meet the requirements and usage access patterns of various workloads.

In KFS, each resource instance (e.g., a particular file, open file instance, block allocation map) is implemented by combining a set of Clustered Objects. The object-oriented nature of KFS makes it a particularly good platform for exploring fine-grained customization. There is no imposition of global policies or data structures.

Several service implementations, with alternative data structuring and policies, are available in KFS. Each element (e.g., file, directory, open file instance) in the system can be serviced by the implementation that best fits its requirements. Further, if the requirements change, the component representing the element in KFS can be replaced accordingly using hot swapping. Applications can achieve better performance by using the services that match their access patterns and synchronization requirements.

In traditional Unix file systems, the inode location is fixed, that is, given an inode number, it is known where to go on the disk to retrieve or update its persistent representation. In KFS, the data representation format for a given inode may vary during its lifetime, potentially requiring a new location on disk. As in the Log-Structured File system (LFS) [29], inode location is not fixed. The *RecordMap* object maintains the inode location mapping. It also keeps information about the type of implementation being used to represent the element, so that when the file/directory is retrieved from disk, the appropriate object is instantiated to represent it.

9. PERFORMANCE MONITORING

The importance of using performance monitoring to achieve good multiprocessor performance is generally understood, especially for multiprocessors. With K42 we have shown the usefulness of a performance monitoring infrastructure [39] well-integrated into the system. Many operating systems did not contain in their original design a mechanism to understand performance. Many times, as those systems evolved, different tailored mechanisms were implemented to examine the portion of the system that needed evaluation. For example, in Linux, there's a device-driver tracing infrastructure, one specific to the file system, the NPTL trace facility, one-off solutions by kernel developers for their own code, plus more-general packages including oprofile, LKST, and LTT. Even commercial operating systems often have several different mechanisms for obtaining tracing information, for example, IRIX [32] had three separate mechanisms. Other mechanisms such Sun's DTrace provide good dynamic functionality but are not well suited for gathering large amounts of data efficiently. Some of these mechanisms were efficient, but often they were one-off solutions suited to a particular subsystem and were not integrated across all subsystems.

Part of the difficulty in achieving a common and efficient infrastructure is that there are competing demands placed on the tracing facility. In addition to integrating the tracing infrastructure with the initial system design, we have de-

veloped a novel set of mechanisms and infrastructure that enables us to use a single facility for correctness debugging, performance debugging, and performance monitoring of the system. The key aspects of this infrastructure are the ability to log variable-length events in per-processor buffers without locks using atomic operations, and given those variable-length events, the ability to randomly access the data stream. The infrastructure, when enabled, is low impact enough to be used without significant perturbation, and when disabled has almost no impact on the system, allowing it to remain always ready to be enabled dynamically.

K42's infrastructure provides cheap and parallel logging of events across applications, libraries, servers, and the kernel into a unified buffer. This event log may be examined while the system is running, written out to disk, or streamed over the network. The unified model allows understanding between different operating system components and across the vertical layers in the execution stack.

Post-processing tools can convert the event log to a human readable form. We have written a set of trace analysis tools, including one designed to graphically display the data. It is hard to overstate the importance of being able to graphically view the data. This capability has allowed us to observe performance difficulties that would otherwise have gone undiagnosed. Another tool evaluates lock contention on an instance by instance basis. In addition, the data can be used for correctness debugging and performance monitoring.

The tracing infrastructure in K42 was designed to meet several goals. The combination of mechanisms and technology we employ achieves the following:

1. Provide a unified set of events for correctness debugging, performance debugging, and performance monitoring.
2. Allow events to be gathered efficiently on a multiprocessor.
3. Efficiently log events from applications, libraries, servers, and the kernel into a unified buffer with monotonically increasing timestamps.
4. Have the infrastructure always compiled into the system allowing data gathering to be dynamically enabled.
5. Separate the collection of events from their analysis.
6. Have minimal impact on the system when tracing is not enabled, and zero impact by providing the ability to "compile out" events if desired.
7. Provide cheap and flexible collection of data for either small or large amounts of data per event.

The tracing facility is well integrated into our code, easily usable and extendable, and efficient, providing little perturbation of the running system. The facility has been invaluable in helping us understand the performance of application and the operating system and in achieving good scalability in K42. Further, because K42 can run Linux application, it has been used by people trying to understand the performance of a Linux application. They have run the application on K42 using K42's performance analysis tools, make improvements, and then can obtain those same improvements when they run again on Linux.

10. LINUX COMPATIBILITY

To be an operating system with the capability of running large applications requires supporting an existing well-known and well-used API. While this requirement of backward compatibility hampers system development and performance, it remains a necessity in today's computing community. In addition to an API, research operating systems face the challenge of supporting enough devices, network protocols, filesystems, and programs to become relevant as a platform. In K42 we have made design decisions to manage this complexity that have resulted in a full-featured user space with underlying support for many important protocols running on a critical subset of PowerPC hardware.

Although K42 supports the standard Linux software stack from the C library and upward, it is an object-oriented kernel and departs significantly from traditional UNIX kernel design. Our design decisions have thus been a balancing act between getting as much from Linux and its associated open source user space as we can, while remaining capable of exploring what we set out to do with our design principles.

For example, in a design inspired by the Exokernel[14] work, processes under K42 can directly branch to certain unprivileged kernel code that is behind a system call gate in most UNIX designs. This poses a conflict with our desire to use the GNU C library (glibc) in unmodified form, as glibc is written to issue a system call instruction in its call stubs. We have settled on an approach where we support glibc in completely unmodified form straight from a standard Linux distribution by reflecting system call traps back into the application's address space, but also provide a patch that one may apply to glibc to replace the traps with direct branches. System call micro-benchmark results indicate that the direct branch is 44% faster than trap reflection.

K42 can be cross-compiled with the standard GNU tool chain, augmented with a stub compiler that allows programmers to express inter-procedure call invocations as normal C++ method calls. K42 is distributed with a root filesystem that provides a standard Linux environment out of the box. For people with an x86 Debian development machine, the setup required to build K42 is to install the Debian development kit packages and build. K42 can then be booted on several different PowerPC hardware platforms, including Apple's G5, POWER3, and POWER4, or on the Mambo full system simulator[8].

K42 accomplishes kernel support for Linux compatibility by directly linking in Linux's TCP/IP stack, filesystems, and device drivers, such as those for disks and network cards. The Linux kernel contains hundreds of thousands of lines of device driver, network, and filesystem code. Unfortunately for research operating systems like L4[26] and K42, this code is tightly coupled and somewhat difficult to reuse. K42 has developed techniques to provide the Linux code with an environment similar to idealized hardware, but even so, a fairly significant maintenance effort is required to keep K42's use of Linux up to date.

Linux application environment

The environment of a K42 user who logs in via ssh or rlogin appears very similar to a regular Linux distribution. Standard utilities like bash, ls, and gcc work as expected, and large applications such as Apache and MySQL work mostly as expected. We have run MPI applications on heterogeneous clusters of Linux and K42 nodes, and on homogeneous

clusters of up to eight K42 nodes. However, one of the core value propositions of K42 is that complex performance sensitive Linux applications can be iteratively accelerated by selectively rewriting critical sections to directly branch to K42 services rather than invoking Linux system calls. To support this, K42 offers macros that an application programmer can use to switch the application in and out of Linux emulation mode.

As described, we allow unmodified Linux binaries and libraries to generate system-call exceptions, which the K42 kernel reflects to the system library for execution. The process is typically referred to as trap reflection.

We provide an alternative glibc library that has a different implementation of the system-call mechanisms that, instead of generating exceptions, calls into the K42 system library directly. This option relies on the system library publishing a vector of system-call entry points at a known location so that the system-call mechanism does not rely on referencing symbols in the K42 system library. The reason for this is that the K42 system library should be invisible to applications that have not been linked against it. Furthermore, the K42 system library is actually responsible for loading the application and dynamic-linker, and the dynamic-linker cannot have symbol dependencies on an external library.

This approach works well as most applications are written to the glibc interfaces and tolerate the library being changed, as would be the case if a user updated the glibc package on their Linux system.

A design goal to support the majority of applications' requirements was to take advantage of the commonly provided interfaces, but where it improves performance, to call K42 services directly by linking against the K42 system library. This is accomplished by providing a specially modified copy of the dynamic library at run-time; when the dynamic linker loads the library, the ELF headers of the library force the linker to look at the image of the system library that is already in memory, and resolve K42 symbols to that library image.

One major problem with all of these approaches is multithreading/pthreads. K42 implements its own threading schemes that are active when running in the K42 system library. When an application is running a pthread, we must ensure that the thread-specific data associated with that thread is what the pthreads library expects. But K42's thread-specific data is not the same and so we must track when a thread switches from pthread-mode to K42-mode. A possible solution is to write our own implementation of the pthreads library that cooperates with the K42 system library and its notion of thread structures and data.

11. USING K42

One of the goals of the K42 project has been to be able to run large applications so an evaluation against known benchmarks could be done. For K42, we decided to implement the Linux API. On the positive side, this meant that applications that ran on Linux would be available to run on K42. On the negative side, as stated in previous sections, that meant having to support inefficient interfaces. As previous sections have alluded to, even though K42 supports a Linux API there are still difficulties in making it work seamlessly. K42 also aims at being flexible so it can run across different platforms, support different applications, and be used by different users that have varying constraints.

11.1 Running and debugging applications on K42

The focus of the K42 project is on performance, scalability, and maintainability. Thus, rather than attempting to provide a complete Linux API, we initially provided the commonly used subset of the API's functionality, and then added what was needed to enable specific applications and subsystems that became of interest. For example, in the last year we have added support for 32-bit Linux system calls and for a more complete set of Gentoo [35] libraries and commands.

Because most of the functionality of the Linux API was available on K42, a large number of Linux binaries executed on K42 without any changes. The initial set of functionality allowed us to run applications such as Apache, dbench, most of SPEC SDET, SPECfp, SPECint, UMT2K, an ASCII nuclear transport simulation, and test suites such as LTP (Linux Test Project). However, some libraries such as LAM/MPI as well as two large subsystems, specifically the Java Virtual Machine J9 and the Database DB2, needed some functionality that was lacking. More recently, we have made efforts to enable these complex commercial applications. We now have J9 and LAM/MPI running and are currently working on DB2.

Getting the J9 Java Virtual Machine running on K42 was reasonably straightforward. Only a few additional files in /proc had to be supported. However, sometimes discovering which ones were required took longer than might be desired. The ones that were needed were those that identify the processor type to the just-in-time compiler. The LAM/MPI library required the ssh server and client, and these required that we complete the K42 implementation of Unix-domain sockets, and find an ssh daemon configuration suitable for K42. In addition, ssh uncovered bugs in the K42 implementation of *select*.

DB2, a large commercial database system, now runs on K42. This is an important milestone for a research operating system. To get DB2 working required finishing Unix domain socket implementation, additional /proc functionality, semaphores, and some aspects of shared memory. We extended our ability to tracking missing components to ioctl and filectl. With these large subsystems now in place K42 has become a reasonable stable platform for experimenting with new applications. However, developers still must be aware that missing functionality may turn up when running new applications. Currently, the primary places where such issues arise in matching the exact behavior regarding error conditions.

There are many tools to debug K42 itself, and to help in running new applications on K42. Most important among these is the debugger. Each process contains a GNU debugger (gdb) stub. When a process traps, the stub is bound to a TCP/IP port. We can subsequently connect a remote gdb client to the trapped process. We also can force a process to trap and enable this mechanism. We use the same mechanism to debug the kernel, but in this case we connect the stub to the serial port. Currently gdb does not run natively on K42.

11.2 Development model

A common goal of open-source software is to be available to a wide audience of users. This implies the software should be configurable in different ways to meet the disparate audi-

ences' needs and should be flexible enough so the core code base does not fragment. As a framework for conducting operating systems research, K42 1) allows groups to leverage development work of other groups, 2) allows developers on different hardware platforms to take advantage of the features available on those platforms, 3) runs on both large- and small-scale machines, and 4) provides a mechanism and tools for developers to understand the performance of the system. Though additional requirements could be listed, the overarching theme is that system software should be flexible so it can run across different platforms, support different applications, and be used by different users that have varying constraints. The description of K42 throughout this paper is consistent with these goals and thus forms a successful development model for an open-source community.

The modular design allows rapid prototyping of operating system policies and mechanisms, providing developers a conduit through which they may quickly test potential operating system innovations. The modular design, in combination with the Linux API and ABI support, allows developers interested in new operating system policies, e.g., a new memory management policy, to experiment without requiring detailed knowledge of the whole system. Technologies that are shown to be beneficial can then be incorporated into Linux [28, 40].

Another key consequence of K42's object-oriented structure is that K42's per-instance resource management allows base K42 code to affect only specific applications. Thus, developers with non-mainstream needs can have their code integrated into K42 without affecting other users, unless those users also desire the modifications. This allows developers with more esoteric needs to contribute as effectively as those whose needs are mainstream allowing broad community involvement.

12. VIRTUALIZATION

Since its inception in 1996, K42 has had the notion of *application managers*. An application manager is a copy of K42 configured for the number of processors needed by a given application. Multiple copies of K42 interact via a very thin layer that provided basic multiplexing. Several years ago, a larger research effort was initiated that is now called the Research Hypervisor[36]. The technology that was developed in the Research Hypervisor is being transfered to PowerPC Xen.

Work on the Research Hypervisor follows the para-virtualization model. This model implies slight modifications to the operating system and the awareness that it is running on a hypervisor. With this model, applications can run at the speed the operating system would have provided. The rapid prototyping advantages of K42 provide a good infrastructure for studying how best to optimize the interface between the operating system and the hypervisor.

In K42 we are examining the issue of what is the best level of interface between the operating system and the hypervisor and how much functionality the hypervisor needs to provide. We are also exploring the notion of providing a minimal library operating system on which to run applications. This expands the notion of hardware resources we had designed in application managers and extends it to the software realm as well. The goal we are exploring with K42 and the Research Hypervisor is to determine how to design a library OS providing only the services needed by the application running on that library OS, thereby decreasing OS perturbation, increasing maintainability, and providing an easy path for backward compatibility.

We believe that a hypervisor should be as thin a layer as possible and that the Hypervisor should only be concerned with the management of memory, processors, interrupts, and a few simple transports. The heart of the Research Hypervisor design is to keep the core hypervisor restricted to these items and keep the code as small and simple as possible. All other services are the domain of surrounding cooperating operating systems.

The goals of the Research Hypervisor project are as follows:

- Be a small, auditable, and configurable source base;
- Provide security through complete isolation, and allow attestation;
- Support open-source operating systems such as Linux, BSD, Darwin, etc.;
- Explore architectural and processor enhancements;
- Develop simple library operating systems to run applications;
- Provide support for real-time operating systems;
- Support full virtualization from within an operating system;
- Provide operating system management;
- Provide new logical transports and inter-operating-system services.

13. FINAL REMARKS

In this section, we describe the impact of the changing technology on K42's design and then present concluding remarks.

13.1 Evolution of the system

The prediction about Windows dominating the market did not come to pass. By around 2000, it became clear to the team that Linux had a real chance of competing. The K42 team became heavily involved in IBM's Linux strategy. We contributed to the 64-bit PPC Linux community code (glibc, gdb, ABI, tool chain) that we had developed for K42, and we transitioned K42 from supporting multiple personalities to supporting the Linux personality with the ability to call past this personality to exploit K42 specific features. The decision to not support multiple personalities reduced the importance of moving functionality into application libraries.

The predictions about the importance of massive NUMA multiprocessors have not borne out to date. For large servers, while systems have had some NUMA characteristics, and caches arguably make all systems NUMA, there has been an increasing trend in the industry to solve problems in hardware. That is, rather than design software that can exploit relatively loosely coupled NUMA systems, hardware has been designed to make the large systems look to be as tightly coupled as possible to avoid having to modify the software. The impact of this trend is that large multiprocessors are both smaller than we predicted, and are increasingly less price/performance competitive with small-scale systems. We continue to believe that economics will force more loosely coupled NUMA systems to become important, and the investments we have made in supporting such systems will be important.

The predictions we made that chips would be multi-core and multi-threaded did happen as supported by recent announcements by IBM, Intel, and AMD. However, this trend occurred more slowly than expected. This meant that the

investments we made in scalability have not until recently been of relevance except to high-end servers. However, the technologies developed in K42 for handling multiple cores are becoming of more interest for low-end machines.

Our prediction about the cost of innovating in existing operating system has, we believe, been correct. The pace of evolution of operating systems, the cost of making incremental modifications, and the massive scale of the open-source effort needed for Linux, demonstrate the challenges of current operating system design. Moreover, operating systems, including Linux, have moved incrementally to a more object-oriented design.

The customizability turned out be valuable. In addition to providing customizability within and across applications running simultaneously on the machine, it has allowed us to pursue hot swapping and dynamic upgrade capabilities in the OS. The object-oriented design has been critical to achieving these capabilities.

The prediction about 64-bit processors becoming dominant was, again, optimistic. Only in the last year are 64-bit processors starting to appear in the high-volume low-end market. The optimizations we did that relied on 64-bit architectures have resulted in K42 being available on only more expensive machines, limiting collaborative opportunities (except recently with Apple's Xserve). Moreover, the lack of support for 64-bit processors in open-source toolchains resulted in the group investing a large amount of time and effort in infrastructure in the early years. This infrastructure turned out to be very valuable in getting a 64-bit Linux on the PowerPC architecture, but for our small research group it has been a large time sink. Only in the last year has a larger community started developing around K42, and that community is porting the system to Intel and AMD processors.

One area we only started working on only in 2002 was the application manager. By that time, IBM had adopted a hypervisor as a part of the firmware for IBM's Power system, and it became clear that a hypervisor would not only enable fault containment, but would also allow K42 to coexist with other operating systems. We ended up writing our own hypervisor, IBM's "research hypervisor", and have more recently transitioned our research in this space to the Xen hypervisor for Power from Cambridge.

13.2 Conclusion

The K42 project has been exploring full-scale operating system research for nine years. In this paper, we presented an overview of the directions we have pursued in K42. We have successfully demonstrated good performance. We have successfully achieved the goals of scalability and customizability through the object-oriented design. Over time, as more groups contribute, we will examine our other goal of maintainability.

Currently, K42's modular structure makes it a valuable research, teaching, and prototyping vehicle, and we expect that the policies and implementations studied in this framework will continue to be transferred into Linux and other systems. In the longer term we believe it will be important to integrate the fundamental technologies we are studying into operating systems.

Our system is available open source under an LGPL license. Please see our home page `www.research.ibm.com/K42` for additional papers on K42 or to participate in this research project.

Acknowledgments

A kernel development project is a massive undertaking, and without the efforts of many people, K42 would not be in the state it is today. In addition to the authors, the following people have contributed much appreciated work to K42: Reza Azimi, Andrew Baumann, Michael Britvan, Chris Colohan, Phillipe DeBacker, Khaled Elmeleegy, David Edelsohn, Raymond Fingas, Hubertus Franke, Ben Gamsa, Garth Goodson, Darcie Gurley, Kevin Hui, Jeremy Kerr, Edgar Leon, Craig MacDonald, Iulian Neamtiu, Michael Peter, Jim Peterson, Eduardo Pinheiro, Bala Seshasayee, Rick Simpson, Livio Soares, Craig Soules, David Tam, Manu Thambi, Nathan Thomas, Gerard Tse, Timothy Vail, and Chris Yeoh.

14. REFERENCES

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 1995. ACM Press.

[2] J. Appavoo, M. Auslander, D. Edelsohn, D. da Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Providing a Linux API on the scalable K42 kernel. In *Freenix track, USENIX Technical Conference*, pages 323–336, San Antonio, TX, June 9-14 2003.

[3] J. Appavoo, K. Hui, M. Stumm, R. Wisniewski, D. da Silva, O. Krieger, and C. Soules. An infrastructure for multiprocessor run-time adaptation. In *WOSS - Workshop on Self-Healing Systems*, pages 3–8, 2002.

[4] A. Baumann, J. Appavoo, D. da Silva, O. Krieger, and R. W. Wisniewski. Improving operating system availability with dynamic update. In *Workshop of Operating System and Architectural Support for the On-demand IT Infrastructure (OASIS)*, pages 21–27, Boston Massachusetts, October 9, 2004 2004.

[5] A. Baumann, J. Appavoo, D. D. Silva, J. Kerr, O. Krieger, and R. W. Wisniewski. Providing dynamic update in an operating system. In *USENIX Technical Conference*, pages 279–291, Anaheim, CA, April 2005.

[6] A. Baumann, J. Kerr, J. Appavoo, D. D. Silva, O. Krieger, and R. W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *Proc. of 6th Linux.conf.au (LCA)*, Canberra, April 2005.

[7] B. N. Bershad, S. Savage, P. Pardyn, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *ACM Symposium on Operating System Principles*, 3–6 December 1995.

[8] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: a full system simulator for the PowerPC architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):8–12, 2004.

[9] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In *USENIX Technical Conference*, New Orleans, LA, June 1998.

[10] R. Bryant, J. Hawkes, and J. Steiner. Scaling Linux to the extreme: from 64 to 512 processors. In *Ottawa Linux Symposium*. Linux Symposium, 2004.

[11] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosio, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 12–25, 1995.

[12] D. da Silva, L. Soares, and O. Krieger. KFS: Exploring flexilibity in file system design. In *Proc. of the Brazilian Workshop in Operating Systems*, Salvador, Brazil, August 2004.

[13] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. Avm: application-level virtual memory. In *Hot Topics in Operating Systems*, pages 72–77. IEEE Computer Society, May 1995.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *ACM Symposium on Operating System Principles*, pages 251–266, 3–6 December 1995.

[15] M. E. Fiuczynski and B. N. Bershad. An extensible protocol architecture for application-specific networking. In *USENIX. 1996 Annual Technical Conference*, pages 55–64. USENIX. Assoc., 1996.

[16] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: a substrate for kernel and language research. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, New York, NY, USA, 1997. ACM Press.

[17] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation*, pages 87–100, February 22-25 1999.

[18] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems*, 20(1):49–83, February 2002.

[19] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 18(3):229–262, 2000.

[20] R. Grindley, T. Abdelrahman, S. Brown, S. Caranci, D. DeVries, B. Gamsa, A. Grbic, M. Gusat, R. Ho, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, P. McHardy, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic. The NUMAchine multiprocessor. In *Proc. of International Conference on Parallel Processing (ICPP'00)*, pages 487–496. IEEE Computer Society, 2000.

[21] Guardian digital, inc., http://www.guardiandigital.com/.

[22] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad new OS research: Challenges and opportunities. In *Proc. of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005. USENIX.

[23] F. L. R. III. Experience with the development of a microkernel-based, multi-server operating system. In *HotOS - Workshop on Hot Topics in Operating Systems*, pages 2–7, 1997.

[24] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[25] I. M. Leslie, R. B. D. McAuley, T. Roscoe, P. Barham, D. Evers, and R. F. E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas In Communications*, 17(7), May 2005.

[26] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250. ACM Press, 1995.

[27] P. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read Copy Update. In *OLS: Ottawa Linux Symposium*, July 2001.

[28] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 338–367, 26–29 June 2002.

[29] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[30] Security-enhanced linux, http://www.nsa.gov/selinux/index.html.

[31] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. An introduction to the architecture of the VINO kernel. Technical report, Harvard University, 1994.

[32] SGI. Sgi irix. http://www.sgi.com/developers/technology/irix/.

[33] L. Soares, O. Krieger, and D. D. Silva. Meta-data snapshotting: A simple mechanism for file system consistency. In *SNAPI'03 (International Workshop on Storage Network Architecture and Parallel I/O)*, pages 41–52, 2003.

[34] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *USENIX Technical Conference*, pages 141–154, San Antonio, TX, June 9-14 2003.

[35] G. Team. Gentoo linux. http://www.gentoo.org.

[36] R. H. Team. The research hypervisor. www.research.ibm.com/hypervisor, march 2005.

[37] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

[38] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. Ashs: application-specific handlers for high-performance messaging. In *ACM SIGCOMM Conference*, August 1996.

[39] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing*, Phoenix Arizona, November 17-21 2003.

[40] T. Zanussi, K. Yaghmour, R. W. Wisniewski, M. Dagenais, and R. Moore. An efficient unified approach for trasmitting data from kernel to user space. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 519–531, July 23-26 2003.