Enabling autonomic behavior in systems software with hot swapping

by J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, J. Xenidis

Autonomic computing systems are designed to be self-diagnosing and self-healing, such that they detect performance and correctness problems, identify their causes, and react accordingly. These abilities can improve performance, availability, and security, while simultaneously reducing the effort and skills required of system administrators. One way that systems can support these abilities is by allowing monitoring code, diagnostic code, and function implementations to be dynamically inserted and removed in live systems. This "hot swapping" avoids the requisite prescience and additional complexity inherent in creating systems that have all possible configurations built in ahead of time. For already-complex pieces of code such as operating systems, hot swapping provides a simpler, higher-performance, and more maintainable method of achieving autonomic behavior. In this paper, we discuss hot swapping as a technique for enabling autonomic computing in systems software. First, we discuss its advantages and describe the required system structure. Next, we describe K42, a research operating system that explicitly supports interposition and replacement of active operating system code. Last, we describe the infrastructure of K42 for hot swapping and several instances of its use demonstrating autonomic behavior.

As computer systems become more complex, they become more difficult to administer properly. Special training is needed to configure and maintain modern systems, and this complexity continues to increase. Autonomic computing systems address this problem by managing themselves.[1] Ideal autonomic systems just work, configuring and tuning themselves as needed.

Central to autonomic computing is the ability of a system to identify problems and to reconfigure itself in order to address them. In this paper, we investigate hot swapping as a technology that can be used to address systems software's autonomic requirements. Hot swapping is accomplished either by *interpositioning* of code, or by *replacement* of code. Interpositioning involves inserting a new component between two existing ones. This allows us, for example, to enable more detailed monitoring when problems occur, while minimizing run-time costs when the system is performing acceptably. Replacement allows an active component to be switched with a different implementation of that component while the system is running, and while applications continue to use resources managed by that component. As conditions change, upgraded components, better suited to the new environment, dynamically replace the ones currently active.

©Copyright 2003 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

0018-8670/03/$5.00 © 2003 IBM

Hot swapping makes downloading of code more powerful. New algorithms and monitoring code can be added to a running system and employed without disruption. Thus, system developers do not need to be prescient about the state that needs to be monitored or the alternative algorithms that need to be available. More importantly, new implementations that fix bugs or security holes can be introduced in a running system.

The rest of the paper is organized as follows. The next section describes how hot swapping can facilitate the autonomic features of systems software. An important goal of autonomic systems software is achieving good performance. The section "Autonomically improving performance" illustrates how hot swapping can autonomically improve performance using examples from our K42[2] research operating system (OS) as well as from the broader literature. The section that follows describes a generic infrastructure for hot swapping and contrasts it with the adaptive code alternative. Then the section "Hot swapping in K42" describes the overall K42 structure, presents the implementation of hot swapping in K42, and includes a brief status and a performance evaluation. The next section discusses related work, and the concluding section contains some final comments.

Autonomic features through hot swapping

Autonomic computing encompasses a wide array of technologies and crosses many disciplines. In our work, we focus on systems software. In this section we discuss a set of crucial characteristics of autonomic systems software and describe how hot swapping via interposition and replacement of components can support these autonomic features, as follows.

Performance—The optimal resource-management mechanism and policy depends on the workload. Workloads can vary as an application moves through phases or as applications enter and exit the system. As an example, to obtain good performance in multiprocessor systems, components servicing parallel applications require fundamentally different data structures than those for achieving good performance for sequential applications. However, when a component is created, for example, when a file is opened, it is generally not known how it will be used. With replacement, a component designed for sequential applications can be used initially, and then it can be autonomically switched to one supporting greater concurrency if contention is detected across multiple processors.

System monitoring—Monitoring is required for autonomic systems to be able to detect security threats, performance problems, and so on. However, there is a trade-off between placing extensive monitoring in the system and the performance overhead this entails. With support for interposition, upon detection of a problem by broad-based monitoring, it becomes possible to dynamically insert additional monitoring, tracing, or debugging without incurring overhead when the more extensive code is not needed. In an object-oriented system, where each resource is managed by a different instance of an object, it is possible to garner an additional advantage by monitoring the code managing a specific resource.

Flexibility and maintainability—Autonomic systems must evolve as their environment and workloads change, but must remain easy to administer and maintain. The danger is that additions and enhancements to the system increase complexity, potentially resulting in increased failures and decreased performance. To perform hot swapping, a system needs to be modularized so that individual components may be identified. Although this places a burden on system design, satisfying this constraint yields a more maintainable system. Given a modular structure, hot swapping often allows each policy and option to be implemented as a separate, independent component, with components swapped as needed. This separation of concerns simplifies the overall structure of the system. The modular structure also provides data structures local to the component. It becomes conceivable to rejuvenate software by swapping in a new component (same implementation) to replace the decrepit one. This rejuvenation can be done by discarding the data structures of the old object, then starting from scratch or a known state in the new object.

System availability—Numerous mission-critical systems require five-nines-level (99.999 percent) availability, making software upgrades challenging. Support for hot swapping allows software to be upgraded (i.e., for bug fixes, security patches, new features, performance improvements, etc.) without having to take the system down. Telephony systems, financial transaction systems, and air traffic control systems are a few examples of software systems that are used in mission-critical settings and that would benefit from hot-swappable component support.

Extensibility—As they evolve, autonomic systems must take on tasks not anticipated in their original design. These tasks can be performed by hot-swapped code, using both interposition and dynamic replacement. Interposition can be used to provide existing components with wrappers that extend or modify their interfaces. Thus, these wrappers allow interfaces to be extended without requiring that existing components be rewritten. If more significant changes are required, dynamic replacement can be used to substitute an entirely new object into an existing running system.

Testing—Even in existing relatively inflexible systems, testing is a significant cost that constrains development. Autonomic systems are more complicated, exacerbating this problem. Hot swapping can ease the burden of testing the system. Individual components can be tested by interposing an object to generate input values and examine results, thereby improving code coverage. Delays can be injected into the system at internal interfaces, allowing the system to explore potential race conditions. This concept is motivated by a VLSI (very large scale integration) technique whereby insertion of test probes across the chip allows intermediate values to be examined.[3,4]

Autonomically improving performance

As outlined in the previous section, autonomic computing covers a wide range of goals, one of which is improving performance. For systems software, the ability to self-tune to maintain or improve performance is one of the most important goals. In this section, we discuss how hot swapping can support and extend existing performance enhancements, allowing the OS to tailor itself to a changing environment.

Optimizing for the common case. For many OS resources the common access pattern is simple and can be implemented efficiently. However, the implementation becomes expensive when it has to support all the complex and less common cases. Dynamic replacement allows efficient implementations of common paths to be used when safe, and less-efficient, less-common implementations to be switched in when necessary.

As an example, consider file sharing. Although most applications have exclusive access to their files, on occasion files are shared among a set of applications. In K42, when a file is accessed exclusively by one application, an object in the application's address space handles the file control structures, allowing it to take advantage of mapped file I/O, thereby achieving performance benefits of 40 percent or more.[5] When the file becomes shared, a new object dynamically replaces the old object. This new object communicates with the file system to maintain the control information. Other examples where similar optimizations are possible are (a) a pipe with a single producer and consumer (in which case the implementation of the pipe can use shared memory between the producer and consumer) and (b) network connections that have a single client on the system (in which case data can be shared with zero copy between the network service and the client).

Optimizing for a wide range of file attribute values. Several specialized file system structures have been proposed to optimize file layout and caching for files with different attributes.[6,7] We can optimize the performance across the range of file attribute values by implementing a number of components, where each component is optimized for a given set of file attribute values, and then having the OS hot swap between these components as appropriate.

For example, although the vast majority of files accessed are small (<4 KB), OSs must also support large files as well as files that grow in size. Using dynamic replacement we can take advantage of the file size in order to optimize application performance. In K42, in the case of a small unshared file, the memory contents backing the file are copied to the application's address space. An object in the application's address space services requests to that file, thus reducing the number of interactions between the client and file system. Once a file grows to a larger size, the implementation is dynamically switched to another object that communicates with the file system to satisfy requests. In this case the file is mapped in memory, and misses are handled by the memory management infrastructure.

Access patterns. There is a plethora of literature focused on optimizing caching and prefetching of file blocks and memory pages from disk based on application access patterns.[8,9] Researchers have shown up to 30 percent fewer cache misses by using the appropriate policy. Hot swapping can exploit these policies by interposing monitoring code to track access patterns and then switching between policies based on the current access pattern.

Exploiting architecture features. Many features of modern processors are underutilized in today's mul-

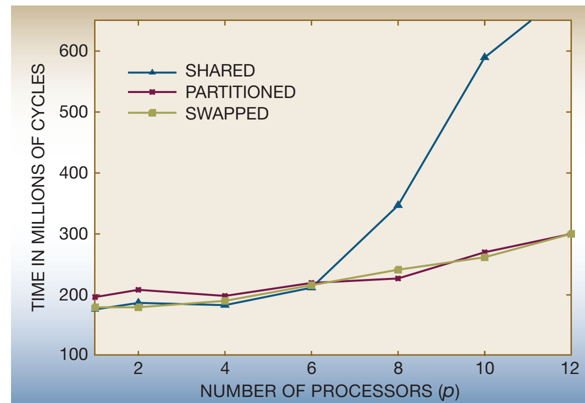
tiplatform OSs. To ensure portable code, without making global code paths unduly complex, these features are generally either crippled or ignored entirely. This is because implementers need to provide a single implementation to be used across all platforms. For example, there is only limited OS support today for large pages, even though a large number of processors support them. Hot swapping makes it easier to take advantage of such architectural features, because special-purpose objects can be introduced and used without requiring that complete functionality for all possible cases be implemented in every object.

Multiprocessor optimizations. In large multiprocessor systems, parallel applications can benefit from processor locality. To exploit this locality, some OSs implement services in a partitioned fashion (code is divided across different processors in order to distribute the load and avoid contention). However, these partitioned implementations consume more memory and incur larger overheads on some operations, for example, file destruction and process destruction. Conversely, shared implementations (code on one processor) can minimize space and time overheads for sequential applications.

Figure 1, which illustrates a file-searching application, can be used to visualize the performance advantages of dynamically switching between a shared and partitioned version of the objects that cache file pages in K42. The system monitors the number of application threads and switches between implementations when appropriate. The *y* axis shows the number of machine cycles to complete a sample search; lower is better. The figure shows that the shared implementation has a 10 percent performance advantage over the partitioned implementation when only one application is searching through the file on one processor. On the other hand, the shared implementation is 300 percent worse with 12 applications on 12 processors. With hot swapping, the system can dynamically switch between the two implementations and capture the best performance characteristics of each.

Enabling client-specific customization. Extensible OSs offer new interfaces that enable clients to customize OS components. By using components optimized for a particular application, it is possible to achieve significant performance improvements in a variety of system services.[10–13] For example, the Exokernel Cheetah Web server demonstrated factor of two-to-four increases in throughput from network

Figure 1 Processor cycles required for *p* independent concurrent searches over an entire 52 MB file (low is good)



stack and file cache specializations.[14] Hot swapping enables extensibility by allowing applications to replace OS components. Hot swapping improves upon most existing extensible systems by allowing on-the-fly switching, as well as replacement of generic system components.

Exporting system structure information. Technologies such as compiler-directed I/O prefetching[15] and storage latency estimation descriptors[16] have shown over 100 percent performance increases for applications, but require detailed knowledge about the state of system structures. Always gathering the necessary profiling information increases overhead and can negatively affect the performance of applications that do not require this information. Hot swapping allows selected applications to gather more information about the state of system structures by interposing a monitoring component when appropriate. By inserting these monitors only when applications will benefit, overall system performance will not degrade. Without hot swapping, the additional cost of monitoring and increased system complexity hampers the ability of researchers to consider algorithms designed for rare conditions that may be important for certain applications.

Supporting specialized workloads. In monolithic systems, optimizations in support of one type of workload often negatively impact the performance of other types of workload. To alleviate this problem some development groups have implemented multiple versions of an OS, where each version is

tuned for a particular type of workload. Another approach is incremental specialization,[17] where specific portions of the kernel are recompiled to optimize them for a particular type of workload. An OS using hot swapping can dynamically switch components optimized to handle these types of workload. The reported performance improvements when using incremental specialization—as high as 70 percent for small amounts of data read from files—can also be obtained using hot swapping.

An infrastructure for hot swapping

Achieving an effective generic hot-swapping mechanism for systems software requires careful design. In addition to the impact on the surrounding system infrastructure that has to be taken into account, there are several actions involved in performing a hot swap, including triggering the hot swap, choosing the target, swapping components, transferring state, and potentially adding object types. In this section, we first describe system requirements for supporting hot swapping, which involve both interposition and replacement, and then we describe the steps involved in performing a component switch. We conclude this section by comparing hot swapping to adaptive code.

System structure. Many large systems, such as databases, are structured with well-defined components and interfaces to those components. This modular structure is critical for hot swapping. Well-defined interfaces are necessary for interposition and replacement of components. Any code, whether it is the kernel, a database, a Web server, or any other server or application at user level, can use the infrastructure for hot swapping. The code intended to perform the hot swap need only be structured so that there are identifiable components that can be interposed or replaced.

In a system with only global components, hot swapping can be used to change overall system performance, but it becomes difficult to tune the system to specific applications because the same component is used across all applications. Additional advantages can be gained if an object-oriented design is used, where each individual use of a resource is managed by an independent object that can be hot swapped in order to tune that resource to its workload. For example, optimization on a per-file basis is possible if each file is implemented using a different object instance that can be tuned to its access pattern.

Large parts of our existing OSs are not designed in a fashion that allows for hot swapping. However, the UNIX** Vnode[18] interface, streams facility, and device driver interface are good examples where hot swapping would be possible.

Modularity and the use of object-oriented design in OSs is expanding. Some current OS interface designs have demonstrated the effectiveness of modularity by enabling flexibility and innovation. For example, there are many Linux** file systems that have explored various possible designs behind the well-defined Linux VFS (virtual file system)[19] interface. As systems become more complex, and autonomic computing becomes more important, the incentives to adopt such designs will grow.

The rest of this paper is presented with an object-oriented structure in mind, and we use the terms "component" and "object" interchangeably. However, much of this discussion applies to systems that are not object-oriented but intend to support hot swapping.

Performing hot swapping. Perhaps surprisingly, only a small number of research groups have looked into hot swapping,[17,20,21] and even then, their approaches have been limited by restrictive conditions. One of the reasons may be the difficulty in providing a general and efficient service that can safely and efficiently handle interposing and replacing components on a multiprocessor in a multithreaded and preemptive environment. For demonstration purposes, consider the difficulties in replacing the TCP/IP (Transmission Control Protocol/Internet Protocol) stack. To do this requires: (1) synchronizing the switching with a potentially large number of concurrent calls from applications and various parts of the OS; (2) atomically transferring state that may include active connections, buffers, and outstanding timers; (3) installing the new object in the system so that its clients automatically and transparently use the new object instance.

The complexity of hot swapping components suggests that the implementer of a specific object will consider providing hot swapping only if the system infrastructure minimizes the implementation work needed in the individual component. Below we discuss a framework that accomplishes this, and in later sections we describe how we have implemented the infrastructure in K42.

Triggering hot swapping. In many cases we expect an object itself to trigger a replacement. For example, if an object is designed to support small files and it registers an increase in file size, then the object can trigger a hot swap with an object that supports large files. In other cases, we expect the system infrastructure to determine the need for an object replacement through a hot swap. Monitoring is required for this purpose, and additional monitoring can be enabled by object interposition if more accurate information is needed before initiating the swap. For example, an OS might have a base level of monitoring in order to identify excessive paging. When this condition occurs the OS might interpose additional monitoring on objects that cache files in order to determine the source of the problem before choosing a specific object instance to replace.

In some cases, applications will explicitly request an object swap. Subsystems such as databases, Web servers, or performance-sensitive scientific applications, can choose to optimize their performance by explicitly switching in new system objects to support known demands. For example, a database application may request the system use objects that support large pages for the purpose of backing a specific region of memory.

In the future, we expect that developers of autonomic computing systems will provide the service infrastructure that allows their products to query for the latest changes, such as bug fixes and security upgrades (similar to the up2date program in Red Hat Linux 7.3). These systems will periodically download new components and hot swap them in without disrupting running applications.

Choosing the target. In some cases, the initiator of a hot swap can identify the target directly as, for example, when upgrading a component. In most cases, however, the target component is more appropriately identified by its behavior than by its specific name or type. For example, a client might request a page-caching object optimized for streaming without needing to know the particular class that implements that functionality. Although introducing such a facility is relatively simple, the complexity comes both in identifying the characteristics that it should encode and the presentation of the encoding to the requester.

Performing the swap. In our experience, the most complex part of hot swapping is performing the swap, including getting the object in a state suitable for swapping and performing the swap in a scalable manner (across a large number of processors). The synchronization needed to get into such a state is complex and not recommended to be implemented on a case-by-case basis. Moreover, synchronization internal to an object imposes a run-time overhead even when hot swapping is not required. In the next section we discuss the implementation of a generic hot-swap mechanism in K42.

Transferring state. A key issue is how to transfer state efficiently between the source and the target of an object replacement. In many cases, the transfer of state between objects being switched is simple. For example, in K42 when an application-level object that caches file state is swapped, we invalidate cached state and pass only the control information. In other cases, the work is more involved. For example, file caching objects convert their internal list of pages into a list of generic page descriptors.

The infrastructure cannot determine what state must be transferred. It can, however, provide a means to negotiate a *best common format* that both objects support. Rather than flattening everything to a canonical state, in some cases pointers to complex data structures may be passed. This is best worked out for a given class hierarchy of objects. Additionally, on a multiprocessor, the infrastructure allows the transfer to occur in parallel.

Dynamically adding object types. Downloading new code into the OS provides two challenges that need to be handled by the infrastructure. First, if an object class is changed, it is necessary to track all instances of that class in order to be able to replace those. Second, if library or subsystem code is changed, it is necessary to download code into all running applications and subsystems using that library.

Adaptive code alternative. Among other features, hot swapping allows system software to react to a changing environment. A more common approach in systems software to handling varying environments is to use adaptive code. Although adaptive code may not achieve the full autonomic vision previously outlined, a comparison to hot swapping is pertinent. In the simplest case, adaptive code has run-time parameters that can be modified on line. In other cases, there are multiple implementations with different data structures, and the best choice of the implementation to use can vary over time.[8,22,23] Adaptive code is a combination of several individual algorithms, each designed for a particular workload.

Table 1 Realization of autonomic features with adaptive code vs hot swapping

Feature	Adaptive Code	Hot-Swappable Code
Set of possible configurations	Preprogrammed	Can be extended
What gets monitored	Preprogrammed	Can be extended
When monitoring code is in system	Always	Dynamic
Adaptation decision algorithm	Preprogrammed	Can be swapped
Code complexity	Made worse	Reduced
Infrastructure required	None	Significant, but once
Enables on-line patches	No	Yes

Table 1 compares adaptive code to hot swapping. When used for optimizing performance, the solution involving adaptive code has three major disadvantages: required foreknowledge, higher code complexity, and higher overhead.

Adaptive code allows the system to switch between a set of preprogrammed algorithms. The set of available algorithms, the monitoring code used to gather data, and the decision-making code cannot be changed once the system is running.

Adaptive code designed for different situations, or to support many applications, is complex. This is especially true for system code designed to run across a variety of hardware platforms. Coordinating adaptation across the many components is more complicated than allowing each component to make its own adaptation decisions.

Adaptive code lacks interposition capability, and thus imposes some monitoring overhead on all requests. Achieving the right level of monitoring through both stable periods and highly loaded, unstable periods, is a challenge.

A more fundamental limitation of the adaptive code solution is its unsuitability for the larger vision of autonomic computing. For example, without the ability to add new code to the system, it does not provide a mechanism to deal with security upgrades or bug fixes. It is possible to update the code off line and restart the system, but this incurs downtime and is disruptive to applications and users.

On a case-by-case basis, the infrastructure for hot swapping we have described is more expensive than simply using specialized adaptive code. However, it only needs to be implemented once, at system development time. In contrast, adaptive systems typically have to reimplement that complexity in each of the services providing the adaptation.

Figure 2 illustrates two implementations of the same function. The adaptive code approach is monolithic and includes monitoring code that collects the data needed by the adaptive algorithm to choose a particular code path. Algorithm options must be part of the original code, and the code's overall size and complexity are increased. With hot swapping, each algorithm is implemented independently (resulting in reduced complexity per component), and is hot swapped in when needed. Monitoring code can also be interposed as needed. Decision code is decoupled from the implementation of the components. The shared code, for tracking usage patterns (not used in the random case), needs to be integrated into the code paths in the adaptive case and is inherited into each object in the hot-swapping case.

Hot swapping in K42

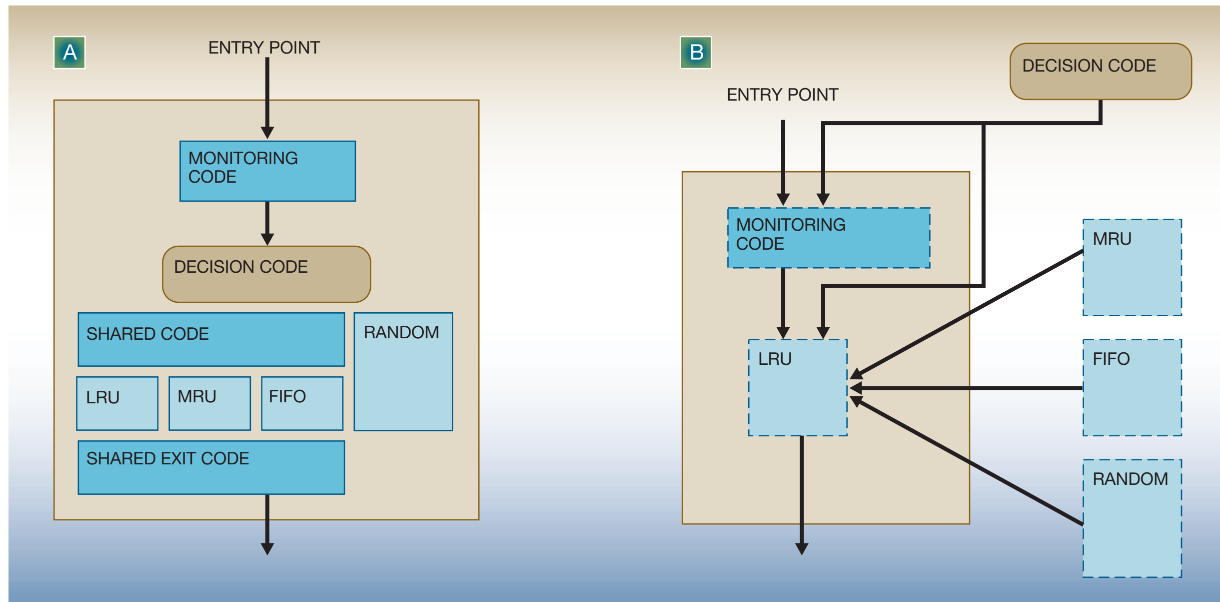
In this section, we describe the generic hot-swapping mechanism of K42. To provide context, we start by presenting the overall structure and design of K42.[2] We then describe K42's infrastructure for hot swapping, give its current status, and present some performance results.

K42. K42 is an open source research kernel for cache-coherent 64-bit multiprocessor systems, which currently runs on PowerPC* and MIPS** platforms, and will soon be available for x86-64 platforms.

Project motivation and goals. K42 focuses on achieving good performance and scalability, on providing a customizable and maintainable system, on supporting a wide variety of platforms, systems, and problem domains, and on being accessible to a large community as Open Source Software.

- *Performance*—K42 is designed to scale up to run well on large multiprocessors and support large-scale applications efficiently. It also scales down to run well on small multiprocessors. Moreover,

Figure 2 An adaptive code implementation (A) vs a hot-swapping implementation (B) of the same function



it supports small-scale applications as efficiently on large multiprocessors as on small multiprocessors.

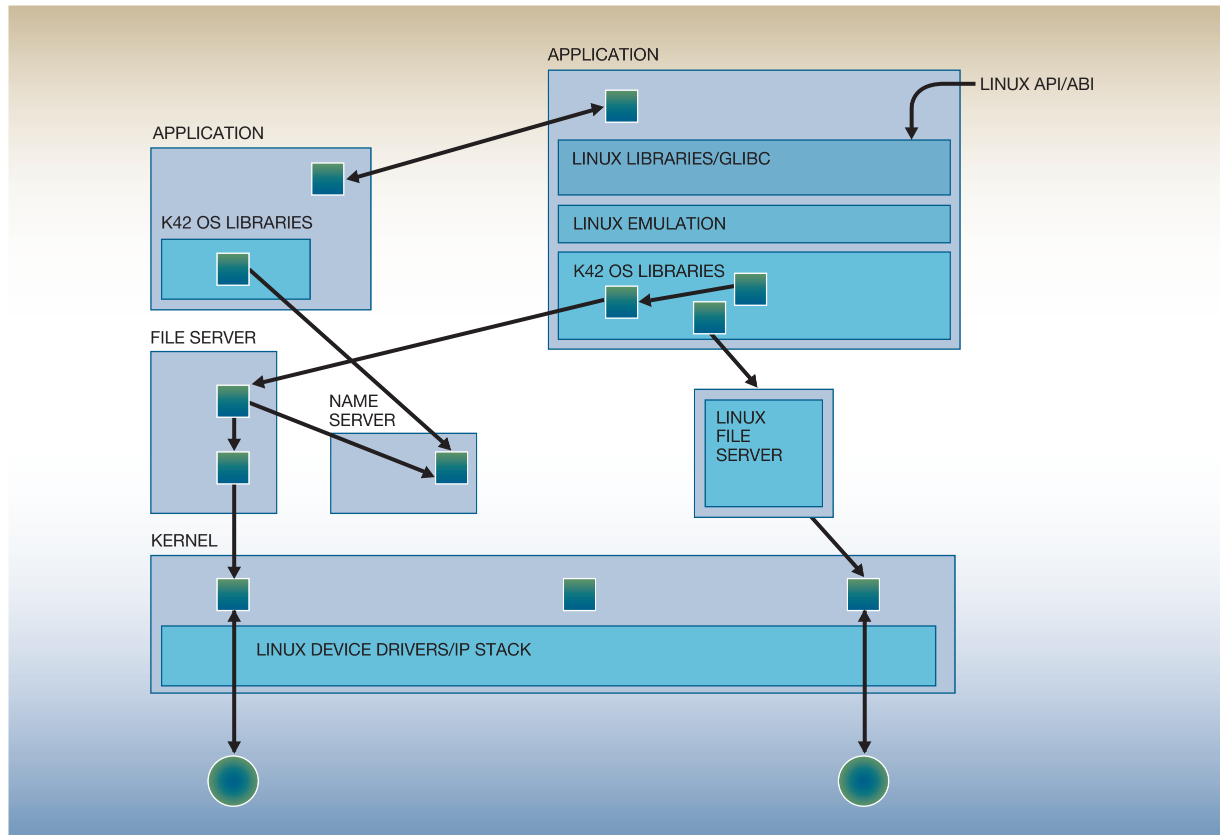
- *Customizability*—K42 allows applications to determine (by choosing from existing components or by writing new ones) how the OS manages their resources. It autonomically adapts to changing workload characteristics.
- *Applicability*—K42 is intended to effectively support a wide variety of systems and problem domains and to make it easy to modify the OS to support new processor and system architectures. It can support systems ranging from embedded processors to high-end enterprise servers.
- *Wide availability*—K42 is open source, and is available to a large community. It makes it easy to add specialized components for experimenting with policies and implementation strategies, and will open up for experimentation parts of the system that were traditionally accessible only to experts.

K42 fully supports the Linux API (application programming interface) and ABI (application binary interface) and uses Linux libraries, device drivers, file systems, and other code without modification. The system is fully functional for 64-bit applications, and can run codes ranging from scientific applications to complex benchmarks like SPEC SDET (Standard Performance Evaluation Corporation Software Development Environment Throughput) to significant subsystems like Apache Software Foundation's Apache HTTP Server. Supporting the Linux API and ABI makes K42 available to a wide base of application programmers, and our modular structure makes the system accessible to the community of developers who wish to experiment with kernel innovations.

Some research OS projects have taken particular philosophies and have followed them rigorously to extremes in order to fully examine their implications. Although we follow a set of design principles, we are willing to make compromises for the sake of performance. The principles that guide our design include (1) structuring the system using modular, object-oriented code, (2) designing the system to scale to very large shared-memory multiprocessors, (3) leveraging performance advantages of 64-bit processors, (4) avoiding centralized code paths, global data structures, and global locks, (5) moving system functionality to application libraries, and (6) moving system functionality from the kernel to server processes.

K42 structure. K42 is structured around a client-server model (see Figure 3). The kernel is one of the core servers, currently providing memory management, process management, IPC (interprocess communication) infrastructure, base scheduling, net-

Figure 3 Structural overview of K42



working, device support, and so on. (In the future we plan to move networking and device support into user-mode servers.)

Above the kernel are applications and system servers, including the NFS (Network File System) file server, name server, socket server, pty (pseudoteletype) server, and pipe server. For flexibility, and to avoid IPC overhead, we implement as much functionality as possible in application-level libraries. For example, all thread scheduling is done by a user-level scheduler linked into each process.

All layers of K42, the kernel, system servers, and user-level libraries, make extensive use of object-oriented technology. All interprocess communication is between objects in the client and server address spaces. We use a *stub compiler* with decorations on the C++ class declarations to automatically generate IPC calls from a client to a server, and have optimized these IPC paths to have good performance. The kernel provides the basic IPC transport and attaches sufficient information for the server to provide authentication on those calls.

From an application's perspective, K42 supports the Linux API and ABI. This is accomplished by an emulation layer that implements Linux system calls by method invocations on K42 objects. When writing an application to run on K42, it is possible to program to the Linux API or directly to the native K42 interfaces. All applications, including servers, are free to reach past the Linux interfaces and call the K42 interfaces directly. Programming against the native interfaces allows the application to take advantage of K42 optimizations. The translation of standard Linux system calls is done by intercepting glibc (GNU C library) system calls and implementing them with K42 code. Although Linux is the first and currently

only personality we support, the base facilities of K42 are designed to be personality-independent.

We also support a Linux-kernel *internal personality*. K42 has a set of libraries that will allow Linux-kernel components such as device drivers, file systems, and network protocols to run inside the kernel or in user mode. These libraries provide the run-time environment that Linux-kernel components expect. This infrastructure allows K42 to use the large code base of hardware drivers available for Linux.

K42 key technologies. To achieve the above mentioned goals, we have incorporated many technologies into K42. We have written several white papers (available on our Web site [2]) describing these technologies in greater detail; the intent of this section is to provide an overview of the key technologies used in K42. Many of these have an impact on hot swapping; for example moving functionality to application libraries makes the hot swapping infrastructure more complicated, but provides additional possibilities for customization and thus for hot swapping. K42 key technologies are listed below. The technologies used by the hot-swapping infrastructure are discussed in "K42 features used" of the subsection "Infrastructure for hot swapping."

- Object-oriented technology, which has been used in the design of the entire system, helps achieve good performance through customization, helps achieve good multiprocessing performance by increasing locality, helps increase maintainability by isolating modifications, and helps perform autonomic functions by allowing components to be hot swapped.
- Much traditional kernel functionality is implemented in libraries in the application's own address space, providing a large degree of customizability and reducing overhead by avoiding crossing address space boundaries to invoke system services.
- K42 is easily ported to new hardware and due to its structure can exploit machine-specific features such as the PowerPC inverted page table and the MIPS software-controlled TLB (translation lookaside buffer).
- Much system functionality has been implemented in user-level servers with good performance maintained via efficient IPCs similar to L4. [24]
- K42 uses *processor-specific* memory (the same virtual address on different processors maps to different physical addresses) to achieve good scalable NUMA (nonuniform memory access) performance. This technology, and avoiding global data, global code paths, and global locks, allows K42's design to scale to thousands of processors.
- Built on K42's object-oriented structure, *clustered objects* [25] provide an infrastructure to implement scalable services with the degree of distribution transparent to the client. This also facilitates autonomic multiprocessor computing, as K42 can dynamically swap between uniprocessor and multiprocessor clustered objects.
- K42 is designed to run on 64-bit architectures and we have taken advantage of 64 bits to make performance gains by, for example, using large virtually sparse arrays rather than hash tables.
- K42 is fully preemptable and most of the kernel data structures are pageable.
- K42 is designed to support a simultaneous mix of time-shared, real-time, and fine-grained gang-scheduled applications.
- K42 has developed deferred object deletion [25] similar to RCU, [26] in which objects release their locks before calling other objects. This efficient programming model is crucial for multiprocessor performance and is similar to type-safe memory. [27]

K42 overall status. K42 is available under an LGPL [28] license (a source tree is available at the URL in Reference 2). We are actively working on providing a complete environment including build and debug tools, simulator, and source. The modular structure of the system makes it a good teaching, research, and prototyping vehicle. Some of the policies and implementations studied in this framework have been transferred into "vanilla" Linux, and we expect that work to continue. Also, in the long term, we expect that the kind of base technologies we are exploring with K42 will be important to Linux.

K42 currently runs on 64-bit MIPS (NUMA) and PowerPC (SMP—symmetric multiprocessor) platforms and is being ported to x86-64. As stated, K42 is fully functional for 64-bit applications, and can run codes ranging from scientific applications to complex benchmarks like SDET, to significant subsystems like Apache. We have demonstrated better base performance for real applications and demonstrated better scalability than other commercial OSs. We expect in the near future to achieve full self-hosting and demonstrate that specialized subsystems can customize the OS to achieve better performance at reduced complexity. There are still edge conditions that have not yet been addressed, and there are still objects with only simplistic implementations.

Infrastructure for hot swapping. In K42, each virtual resource instance, for example, open file instance or memory region, is implemented by combining a set of (C++) object instances we call building blocks.[29] Each building block implements a particular abstraction or policy and might (1) manage some part of the virtual resource, (2) manage some of the physical resources backing the virtual resource, or (3) manage the flow of control through the building blocks. For example, there is no global page cache in K42; instead, for each file there is an independent object that caches the blocks of that file.

K42's infrastructure allows any object to replace any other object implementing the same interface, or to interpose any object with one providing the same interface. In K42 we use hot swapping to replace kernel objects as well as objects in user-level servers. The hot swapping occurs transparently to the clients of the component and no support or code changes are needed in the clients.

Dynamic replacement algorithm: Design issues. This algorithm contains the following steps: (1) instantiate the replacement component, (2) establish a quiescent state for the component to be replaced, (3) transfer state from the old component to the new component, (4) swap in the new component replacing all references, and (5) deallocate the old component.

There are three key issues that need to be addressed in this design. First, we need to establish a quiescent state so that it is safe to transfer state and swap references. The swap can only be done when the component state is not currently being accessed by any thread in the system. Perhaps the most straightforward way to achieve a quiescent state would be to require all clients of the component to acquire a reader-writer lock in read mode before any call to the component. Acquiring this external lock in write mode would thus establish that the component is safe for swapping. However, this would add overhead for the common case, and cause locality problems in the case of multiprocessors. Further, the lock could not be part of the component itself.

Second, we need to determine what state needs to be transferred and how to transfer it to the new component safely and efficiently. Although the state could be converted to a canonical, serialized form, this would lose context, be a less efficient transfer protocol, and potentially prevent parallelism when the transfer is occurring on a multiprocessor.

Third, we need to swap all of the references held by the clients of the component so that the references refer to the new component. In a system built around a single, fully typed language, like the Java** language, this could be done using the same infrastructure as used by garbage collection systems. However, this would be prohibitively expensive for a single component switch. An alternative would be to partition a hot-swappable component into a front-end component and a back-end component, where the front-end component is referenced (and invoked) by the component clients, and is used only to forward requests to the back-end component. There would then be only a single reference (in the front-end component) to the back-end component that would need to be changed when a component is swapped, but this adds extra overhead to the common call path.

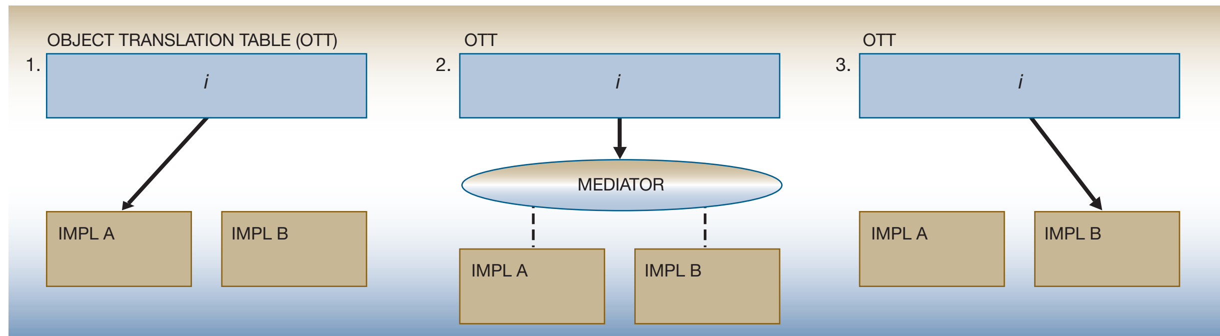
K42 features used. Our implementation of hot swapping leverages three key features of K42 that allow us to address the issues listed above in an efficient and straightforward manner. Similar features exist or could be retrofitted into other systems.

First, because K42 has an object-oriented structure implemented using C++,[29] each system component maps naturally to a language object. Hot swapping is facilitated because the objects are self-contained with well-defined interfaces. A similar approach could be used in a system that is not object-oriented but uses operations tables, such as vnodes.[18]

Second, each K42 object is accessed through a single pointer indirection, where the indirection pointer of all objects is maintained in an Object Translation Table (OTT) indexed by the object identifier. The OTT was originally implemented in K42 to support a new form of scalable structure called *Clustered Objects*.[25] Another method for doing this would be the dynamic linking technology such as that used in ELF (Executable and Linking Format), otherwise the indirection would need to be added explicitly.

Finally, K42 has a *generation count* mechanism that allows us to easily determine when all threads that were started before a given point in time have completed, or reached a safe point. (A similar mechanism has been used by [formerly] Sequent's NUMA-Q**[30] for the same reason we originally developed it for K42, namely to improve multiprocessor performance by deferring expensive, but noncritical operations.[31]) This mechanism is used to achieve a quiescent state for an object. The mechanism exploits the fact that OSs are event-driven, where most

Figure 4 Stages in replacing instance A of object *i* by instance B



requests are serviced and completed quickly. Long-living daemon threads are treated specially. This type of functionality can usually be added to other event-driven systems, such as Web, file, or database servers, in which the thread of control frequently reaches safe points such as the completion of a system call, or entering a sleep.

Description of algorithm. A part of the replacement algorithm involves interpositioning a mediator. Interpositioning is the ability to redirect future calls intended for a given object to another object. To perform interpositioning the object's indirection pointer in the OTT is changed to point to the interposed object. It is not necessary to reach object quiescence. This interposition object remains active and can forward calls to the original object performing whatever operation or monitoring it desires prior to the forwarding.

Conceptually there are three stages in replacing a component as depicted in Figure 4. In the OTT's initial state, the *i*-th table entry contains a pointer to the current (old) object. In the second stage, a mediator is interposed. In the third stage the OTT is in its final state, with the *i*-th entry pointing to the new object.

To perform a replacement, a mediator object is interposed in front of the old object. This mediator object proceeds through the three phases. To establish a quiescent state, in which it is guaranteed that no threads have an active reference to the component to be swapped, the mediator object initially, in the *Forward* phase, tracks all threads making calls to the component and forwards each call on to the original component. It does so until it is certain that all calls started before call tracking began have completed. To detect this, we rely on K42's generation count feature to determine when all calls that were started before tracking began have completed. Until that point, the mediator continues to forward new requests to the original component, which services them as normal.

At that point, the mediator starts the *Blocked* phase and temporarily blocks new calls, while it waits for the tracked calls to complete. Exceptions are made for recursive calls that need to be allowed to proceed to avoid deadlock. If the user subverts the K42 programming model making recursive object calls across servers while simultaneously switching multiple objects, the infrastructure will not be able to detect deadlock loops. To handle this, and potentially other unknown deadlock circumstances, a timeout mechanism is used. If the timeout is triggered, it terminates the hot swap, setting the client pointers back to the original object. In nonerroneous object implementations, calls can be correctly tracked and blocked. Once all the tracked calls have completed, the component is in a quiescent state, and the state transfer can begin. While the *Blocked* phase may seem to unduly reduce the responsiveness of the component, in practice the delay depends only on the number of tracked calls, which are generally short and few in number.

To make state transfer between the original and the new component efficient and preserve as much of the original state and semantics as possible, the original and new objects negotiate a *best common format* that they both support. This, for example, may allow a hash table to be passed directly through a pointer, rather than converted to and from some ca-

nonical form, such as a list or array, as well as, in a large multiprocessor, allow much of the transfer to occur in parallel across multiple processors, preserving locality where possible.

Finally, after the state transfer, the mediator enters the *Completed* phase. It removes its interception by updating the appropriate indirection pointer in the OTT to point to the new component so that future calls go to the new component directly. It also resumes all threads that were suspended during the *Blocked* phase and directs them to the new component. The mediator then deallocates the original object and finally itself.

The implementation of the above design has a number of important features. There is no overhead during normal operation; overhead occurs only when the mediator is interposed, and the mediator is used only during the hot-swapping process. The implementation works efficiently in a multithreaded multiprocessor system. The mediator runs in parallel and state transfer proceeds in parallel. Call interception and mediation is transparent to the clients, facilitated by K42's component system infrastructure. Finally, our solution is generic in that it separates the complexity of swap-time in-flight call tracking and deadlock avoidance from the implementation of the component itself. With the exception of component state transfer, the rest of the swapping process does not require support from the component, simplifying the addition of components that wish to take advantage of the hot-swapping capability.

Status, performance, and continuing work. K42 fully supports the hot-swapping infrastructure described in the previous section. We have used it for simple applications, such as the search program in Figure 1, as well as for complex applications, such as the SPEC SDET benchmark. The code works for all the objects in our system, and the main current limitation is in the number of choices of objects we can hot swap. We are continuing to explore the use of hot swapping in implementing additional autonomic computing features. Next we present a performance study involving a significant application, and then we discuss our plans for the future.

An important aspect of the virtual memory system is keeping track of in-core pages. In K42, this function is implemented by a File Cache Manager (FCM) component. We focus on two of the default implementations: shared and distributed. For each open file, an instance of an FCM is used to cache the pages of the file's data in physical frames. By default, to achieve better performance when a file is opened, a simple shared implementation of the FCM is created. The default decision is made based on the fact that most files are accessed by one thread on one processor and opened only briefly. If the file is accessed on one processor, the shared FCM implementation performs well and has little memory overhead. If the file is accessed by multiple processors concurrently, the associated FCM is hot swapped to a distributed implementation. This alleviates contention and yields better scalability, thus ensuring that only the files that experience contention due to sharing use the more complex and expensive distributed FCM implementation. However, because the shared implementation performs an order of magnitude worse when running on many processors, without hot swapping, an FCM suitable for the distributed case would need to be used all the time and a performance penalty paid in the single processor case.

One of the studies we did to understand the advantages of hot swapping the FCM implementations was to run the PostMark[32] and SPEC SDET benchmarks. PostMark is a uniprocessor file system benchmark that creates a large number of small files on which a number of operations are performed, including reading and appending. SPEC SDET is a multiprocessor UNIX development workload that simulates concurrent users running standard UNIX commands (Due to tool chain issues we used a modified version that does not include a compiler or UNIX command "ps."). If we disable hot swapping and run PostMark using only shared FCMs, and then run it using only distributed FCMs, we find that we suffer a 7 percent drop in performance for the distributed implementation of the FCM. On the other hand, if we run SDET in a similar fashion we find that the distributed FCM gets an 8 percent performance improvement on 4 processors and an order of magnitude improvement on 24 processors over the shared FCM. When hot swapping is enabled, the best performance is achieved automatically for both PostMark and SDET, with the individual FCM instances choosing the right implementation based on the demands it experiences.

The above experiment shows the power of hot swapping. In other work we have studied the performance advantages of hot swapping. The results we have obtained are encouraging, but there is still much to do within the K42 project to bring hot swapping to a mature state. Currently the trigger for an object hot swap is either specified by an object or by an appli-

cation via an explicit request. Although this has proven to be sufficient for many cases, we have found situations where a monitoring infrastructure would allow us to make the trigger decision on behalf of an object. The monitoring code we currently use has often been placed in the object for convenience and to gain experience with how to gather and use the information. We plan to use object interposition to reduce the overhead of this scheme and provide a more generic mechanism for gathering this information.

Determining the target object is done today explicitly by the requester of a hot swap. We are currently extending the K42 type system to provide a service that identifies objects by their characteristics, again allowing a more generic mechanism for handling the hot swap. Because our project has been performance-driven, our focus so far has been to use autonomic computing to improve performance. However, there are important benefits to be gained from being able to autonomically swap in a security patch, or achieve higher availability by a live upgrade. Thus, we are beginning to explore other autonomic features.

There is considerable potential, and required effort, to understand the longer-term and larger issues involved in providing many components that manage a given resource and getting them to safely and correctly interact. So far, we have successfully used hot swapping and its autonomic features in K42 and expect continued progress in our research.

Related work

Although there is a large body of prior work focusing on the downloading and dynamic binding of new components, there has been less work on swapping components in a live system. Hjálmtýsson and Gray describe a mechanism for updating C++ objects in a running program,[21] but their client objects need to be able to recover from broken bindings due to an object swap and retry the operation, so their mechanism is not transparent to client objects. Moreover, they do not detect quiescent state, and old objects continue to service prior calls while the new object begins to service new calls.

The *virtualizing Operating System* (vOS)[33] is a middleware application that offers an abstraction layer between applications and the underlying OS. vOS allows modules to be refreshed, that is, to be dynamically reloaded to achieve software rejuvenation. It does not address loading of new implementations, and its approach does not apply to OS components.

Pu et al. describe a "replugging mechanism" for incremental and optimistic specialization,[17] but they assume there can be at most one thread executing in a swappable module at a time. In later work that constraint is relaxed but is nonscalable. Hicks et al. describe a method for dynamic software updating, but in their approach, all objects of a certain type are updated simultaneously, not just individual instances, as is possible with our scheme.[20] Moreover, they require that the program be coded to decide when a safe point has been reached and initiate the update.

The modular structure, level of indirection, and availability of a mechanism for detecting a quiescent state is not unique to K42. Sequent has a mechanism for detecting quiescent state,[31] and we are working with this group to incorporate a similar facility into Linux. This will allow hot swapping of system modules (i.e., device drivers and file systems).

In general, the work described here can be viewed as part of wide-spread research efforts to make OSs more adaptive and extensible as in SPIN,[10] Exokernel,[11] and VINO.[13] These systems are unable to swap entire components, but rather just provide hooks for customization. Our work is complementary to the above mentioned related work. We focus primarily on a mechanism for swapping generic components in a highly dynamic, multithreaded multiprocessing system. Several groups have also done work on adding extensibility to both applications and systems. CORBA**,[34] DCE,[35] and RMI[36] are all application architectures that allow components to be modified during program execution; but they do not address the performance or complexity concerns present in an OS.

Conclusions

Autonomic systems software should have self-maintenance capabilities, and should be able to run applications well on various hardware platforms and across various environments. We proposed an object-oriented system structure that supports hot swapping as an infrastructure toward meeting these autonomic challenges by dynamically monitoring and modifying the OS. We demonstrated the technical feasibility of the hot-swapping approach by describing our implementation of it in K42. We described how hot swapping can yield performance advantages and

showed an application involving K42. We will continue to explore other autonomic features in K42 by examining security, software upgrading, and multiple object coordination issues.

Our prototype is Open Source Software and available on our Web site.[2] Readers may use the Web site for accessing related white papers, and for contacting us if interested in participating in this project.

Acknowledgments

We would like to thank the anonymous referees for their positive and helpful comments.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of The Open Group, Linus Torvalds, MIPS Technologies, Inc., Sun Microsystems, Inc., Sequent Computer Systems, Inc., or Object Management Group.

Cited references

1. IBM Corporation, Autonomic Computing, http://www.research.ibm.com/autonomic/.
2. IBM Corporation, Research Division, The K42 Project, http://www.research.ibm.com/K42.
3. D. Knebel et al., "Diagnosis and Characterization of Timing-Related Defects by Time-Dependent Light Emission," *Proceedings of the International Test Conference*, October 18–23, 1998, Washington, DC, IEEE, New York (1998), pp. 733–739.
4. M. Paniccia, T. Eiles, V. R. M. Rao, and W. M. Yee, "Novel Optical Probing Technique for Flip Chip Packaged Microprocessors," *Proceedings of the International Test Conference*, October 18–23, 1998, Washington, DC, IEEE, New York (1998), pp. 740–747.
5. O. Krieger, M. Stumm, and R. Unrau, "The Alloc Stream Facility: A Redesign of Application-Level Stream I/O," *IEEE Computer* **27**, No. 3, 75–82 (1994).
6. O. Krieger and M. Stumm, "HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions," *ACM Transactions on Computer Systems* **15**, No. 3, 286–321 (1997).
7. Reiser File System (ReiserFS), see http://www.namesys.com.
8. G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, ACM Press, New York (1997), pp. 115–126.
9. J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A Low-Overhead High-Performance Unified Buffer Management Scheme That Exploits Sequential and Looping References," *Proceedings, Symposium on Operating Systems Design and Implementation*, San Diego, CA, USENIX Association (2000), pp. 119–134.
10. 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," *Proceedings, ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, ACM, New York (1995), pp. 267–283.
11. D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proceedings, ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, ACM, New York (1995), pp. 251–266.
12. D. Mosberger and L. L. Peterson, "Making Paths Explicit in the Scout Operating System," *Proceedings, Symposium on Operating Systems Design and Implementation*, Seattle, WA, ACM, New York (1996), pp. 153–167.
13. M. Seltzer, Y. Endo, C. Small, and K. A. Smith, *An Introduction to the Architecture of the VINO Kernel*, Technical Report TR-34-94, Harvard University, Cambridge, MA (1994).
14. 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**, No. 1, 49–83 (2002).
15. A. D. Brown, T. C. Mowry, and O. Krieger, "Compiler-Based I/O Prefetching for Out-of-Core Applications," *ACM Transactions on Computer Systems* **19**, No. 2, 111–170 (2001).
16. R. V. Meter and M. Gao, "Latency Management in Storage Systems," *Proceedings, Symposium on Operating Systems Design and Implementation*, San Diego, CA, USENIX Association (2000), pp. 103–117.
17. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," *Proceedings, ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, ACM, New York (1995), pp. 314–321.
18. S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of the Summer Conference*, Atlanta, GA, 1986, USENIX Association (1986), pp. 238–247.
19. R. Gooch, "Linux Virtual File System," http://www.atnf.csiro.au/people/rgooch/linux/vfs.txt.
20. M. Hicks, J. T. Moore, and S. Nettles, "Dynamic Software Updating," *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, Snowbird, UT, ACM Press, New York (2001), pp. 13–23.
21. G. Hjálmtýsson and R. Gray, "Dynamic C++ Classes: A Lightweight Mechanism to Update Code in a Running Program," *Proceedings, Annual USENIX Technical Conference*, USENIX Association (1998), pp. 65–76.
22. J. Aman, C. K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger, "Adaptive Algorithms for Managing a Distributed Data Processing Workload," *IBM Systems Journal* **36**, No. 2, 242–283 (1997).
23. Y. Li, S.-M. Tan, Z. Chen, and R. H. Campbell, *Disk Scheduling with Dynamic Request Priorities*, Technical Report, University of Illinois at Urbana-Champaign, IL (August 1995).
24. J. Liedtke, "On Micro-Kernel Construction," *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, ACM Press, New York (1995), pp. 237–250.
25. B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm, "Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System," *Proceedings, Symposium on Operating Systems Design and Implementation*, New Orleans, LA, USENIX Association (1999), pp. 87–100.
26. P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell, "Read Copy Update," *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada (June 2002).

27. M. Greenwald and D. Cheriton, "The Synergy Between Non-Blocking Synchronization and Operating System Structure," *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, ACM Press, New York (1996), pp. 123–136.
28. Free Software Foundation, Inc., GNU Lesser General Public License, http://www.research.ibm.com/K42/LICENSE.html.
29. M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm, "Customization Lite," *Hot Topics in Operating Systems*, IEEE, New York (May 1997), pp. 43–48.
30. T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *Proceedings of the 23rd International Symposium on Computer Architecture*, ACM, New York (1996), pp. 308–317.
31. P. E. McKenney and J. D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems," International Conference on Parallel and Distributed Computing and Systems, Las Vegas, NV, 28–31 October 1998, IASTED (International Association of Science and Technology for Development), Calgary, Canada (1998).
32. J. Katcher, *PostMark: A New File System Benchmark*, TR3022, Network Appliance, Sunnyvale, CA 94089, http://www.netapp.com/tech_library/3022.html.
33. T. Boyd and P. Dasgupta, "Preemptive Module Replacement Using the Virtualizing Operating System," Workshop on Self-Healing, Adaptive and Self-Managed Systems (SHAMAN) (June 2002).
34. Z. Yang and K. Duddy, "CORBA: A Platform for Distributed Object Computing," *ACM Operating Systems Review* **30**, No. 2, 4–31 (1996).
35. *Distributed Computing Environment: Overview*, OSF-DCE-PD-590-1, Open Software Foundation (May 1990).
36. Java Remote Method Invocation—Distributed Computing for Java, Sun Microsystems, Inc., http://java.sun.com/marketing/collateral/javarmi.html.

Accepted for publication July 29, 2002

Jonathan Appavoo *Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4 (electronic mail: jonathan@eecg.toronto.edu).* Mr. Appavoo has a Master of Computer Science degree from the University of Toronto, where he is currently enrolled in the Ph.D. program. He has spent the last four years working on K42 and its predecessor, Tornado. He works closely with the K42 team at the IBM Thomas J. Watson Research Center, where he has interned. His research has focused on multiprocessor operating system performance with a particular interest in scalable data structures.

Kevin Hui *Kaleidescape, Inc., 155 Frobisher Drive, Suite I-205, Waterloo, Ontario, Canada N2V 2E1 (electronic mail: kevin@kaleidescape.com).* Mr. Hui is a software engineer at Kaleidescape, where he works on operating system and information security. Under the supervision of Dr. Michael Stumm, he obtained his M.Sc. in computer science in 2000 at the University of Toronto. Prior to receiving his degree, he enjoyed a productive stay at the IBM Thomas J. Watson Research Center, where he designed and implemented K42's hot-swapping infrastructure with Dr. Robert Wisniewski and the rest of the K42 team.

Craig A. N. Soules *Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania 15212 (electronic mail: soules@cmu.edu).* Mr. Soules has been a graduate student in the Computer Science Department at Carnegie Mellon University since fall 2000. His research has focused mainly on the design and performance of operating systems and file systems.

Robert W. Wisniewski *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: bob@watson.ibm.com).* Dr. Wisniewski is a research scientist at the Watson Research Center working on the K42 operating system, and is currently exploring scalable, portable, and configurable next-generation operating systems. He received his Ph.D. degree in 1996 from the University of Rochester, Rochester, NY, where his thesis was "Achieving High Performance in Parallel Applications via Kernel-Application Interaction." His research interests include scalable parallel systems, first-class system customization, and performance monitoring.

Dilma M. Da Silva *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: dilmasilva@us.ibm.com).* Dr. Da Silva received her B.S. and M.S. degrees in computer science from the University of Sao Paulo, Brazil, and her Ph.D. from Georgia Institute of Technology, Atlanta, GA. From 1996 to 2000 she was an assistant professor in the Department of Computer Science at the University of Sao Paulo, Brazil. She is currently a research staff member at IBM's Thomas J. Watson Research Center. Her research interests include operating systems and dynamic system configuration.

Orran Krieger *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: okrieg@us.ibm.com).* Dr. Krieger is a manager at IBM's Thomas J. Watson Research Center. He received a B.A.Sc. degree from the University of Ottawa in 1985, an M.A.Sc. degree from the University of Toronto in 1989, and a Ph.D. degree from the University of Toronto in 1994, all in electrical and computer engineering. He was one of the main architects and developers of the Hurricane and Tornado operating systems at the University of Toronto, and was heavily involved in the architecture and development of the Hector and NUMAchine shared-memory multiprocessors. Currently, he is project leader of the K42 operating system project at the IBM Research Center and an adjunct associate professor in computer science at Carnegie Mellon University, Pittsburgh, PA. His research interests include operating systems, file systems, and computer architecture.

Marc A. Auslander *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: Marc_Auslander@us.ibm.com).* Mr. Auslander is an IBM Fellow, a member of the National Academy of Engineering, an ACM Fellow, and an IEEE Fellow. He received the A.B. in mathematics from Princeton University in 1963. He joined IBM in 1964 and transferred to the Thomas J. Watson Research Center in 1968. Mr. Auslander contributed significantly to IBM's entry into virtual systems. He was an original member of the group that designed and built the first IBM RISC machine and the first RISC optimizing compiler. He has also made important contributions to AIX®, other operating systems, and the PowerPC architecture. He is currently working on the K42 operating system. His research interests include multiprocessor operating systems, compilers, and processor architecture.

David J. Edelsohn *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: dje@watson.ibm.com).* Dr. Edelsohn is a research

staff member in IBM's Research Division. He received his Ph.D. degree in 1996 from Syracuse University, Syracuse, NY. At Syracuse University's Northeast Parallel Architectures Center, he worked with Geoffrey Fox studying mixed media systems and massively parallel hierarchical algorithms. His research interests include compiler optimizations, operating systems, and parallel computing.

Ben Gamsa *SOMA Networks, Inc., 312 Adelaide Street West, Suite 700, Toronto, Ontario, Canada M5V 1R2 (electronic mail: ben@somanetworks.com).* Dr. Gamsa received his Ph.D. degree in 1999 from the Department of Computer Science at the University of Toronto and is currently employed at SOMA Networks.

Greg R. Ganger *Carnegie Mellon University, Pittsburgh, Pennsylvania 15212 (electronic mail: greg.ganger@cmu.edu).* Dr. Ganger is a professor in the Electrical and Computer Engineering Department at Carnegie Mellon University, Pittsburgh, PA. His broad research interests in computer systems include storage systems, security, and operating systems. He is director of Carnegie Mellon's Parallel Data Lab, academia's premiere storage systems research center. His Ph.D. in computer science and engineering is from the University of Michigan, and he spent two and one half years as a postdoctoral student at Massachusetts Institute of Technology, working on the Exokernel project.

Paul McKenney *IBM Server Group, Linux Technology Center, 15450 S.W. Koll Parkway, Beaverton, Oregon 97006-6096 (electronic mail: pmckenne@us.ibm.com).* Mr. McKenney is a system architect at the Linux Technology Center. Prior to that he worked on locking and parallel operating-system algorithms at Sequent Computer Systems. He is working toward his Ph.D. degree at Oregon Graduate Institute. His research interests include parallelism, virtualization, and autonomic computing.

Michal Ostrowski *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: mostrows@watson.ibm.com).* Mr. Ostrowski is a member of the Advanced Operating Systems (K42) group at the IBM Research Center. He is involved in research and development activities on the K42 operating system, currently focusing on developing a framework for using Linux kernel code in K42 and developing frameworks and mechanisms for efficient asynchronous I/O interfaces. He completed his master's degree at the University of Waterloo in 2000.

Bryan Rosenburg *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: rosnbrg@us.ibm.com).* Dr. Rosenburg received his Ph.D. degree in computer sciences at the University of Wisconsin-Madison in 1986, and immediately thereafter joined the Watson Research Center as a research staff member in the operating system group of the RP3 NUMA multiprocessor project. He has been a member of the K42 team since its inception. His current research interests are in the lowest levels of operating system architecture: interrupt handling, context switching, interprocess communication, and low-level scheduling.

Michael Stumm *Department of Computer Science, University of Toronto, 10 King's College Road, Toronto, Ontario, Canada M5S 3G4 (electronic mail: stumm@eecg.toronto.edu).* Dr. Stumm is a professor in the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto. He received a diploma in mathematics and a Ph.D. degree in computer science from the University of Zurich in 1980 and 1984, respectively. Dr. Stumm's research interests are in the areas of computer systems, in particular operating systems for distributed and parallel systems.

Jimi Xenidis *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: jimix@watson.ibm.com).* Mr. Xenidis is a software engineer at the Watson Research Center working with several groups in addition to the K42 project. His research interests include operating systems, linkers and libraries, and machine virtualization.