

Enabling Autonomic System Software with Hot-Swapping

Jonathan Appavoo[†] Kevin Hui[†] Craig A. N. Soules[‡] Robert W. Wisniewski[§]

Dilma Da Silva[§] Orran Krieger[§] Marc Auslander[§] David Edelsohn[§]

Ben Gamsa[¶] Gregory R. Ganger[‡] Paul McKenney^{||} Michal Ostrowski[§]

Bryan Rosenburg[§] Michael Stumm^{**} Jimi Xenidis[§]

Abstract

Autonomic computing systems are designed to be self-diagnosing and self-modifying, such that they notice performance and correctness problems, pinpoint their causes, and react accordingly. These abilities can increase performance, uptime, and security, while simultaneously reducing the effort and knowledge 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”

[†]University of Toronto, Dept of Computer Science

[‡]Carnegie Mellon University

[§]IBM T. J. Watson Research Center

[¶]SOMA Networks, Inc.

^{||}IBM Beaverton

^{**}Univ of Toronto, Dept of Electrical and Computer Engineering

approach 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.

This paper describes an aspect of autonomic computing for systems software using hot-swapping. We describe its advantages and system structure requisite for hot-swapping. The K42 Operating System (OS) explicitly supports interposition and replacement of active OS code. We describe K42's infrastructure and give examples from our research work demonstrating autonomic behavior.

1 Introduction

As computer systems become more complex, they become more difficult to administer properly. Modern systems are so complex that special training is needed to configure and maintain them, and this complexity is continuing to increase. Autonomic computing systems address this problem by managing themselves. 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 reconfigure itself to account for them. In this paper, we investigate hot swapping as a technology that can be used to address system software's autonomic requirements. Hot swapping is the dynamic insertion and removal of code in running systems. It consists of two pieces: *interpositioning* and *replacement*. *Interpositioning* involves inserting additional components between existing ones. This allows, for example, more detailed monitoring to be enabled when problems occur while al-

lowing the run-time cost to be minimized when the system is performing acceptably. *Replacement* allows an active component to be switched with another component while the system is running, and while applications are using the resource managed by the component. This allows components suited to a particular environment to be switched as conditions change and allows new upgraded components to replace existing ones.

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.

Section 2 describes how hot swapping can facilitate autonomic systems software. An important goal of autonomic systems software is achieving good performance. Section 3 illustrates how hot-swapping can autonomically improve performance with examples from our K42 research operating system as well as from the broader literature. Section 4 describes the system infrastructure required to support hot-swapping. Section 5 describes the overall K42 structure, presents the implementation of hot-swapping in K42, and includes a brief status and performance evaluation. Section 6 discusses related work.

2 Motivation and Goals

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

can provide the needed technology.

Performance: The optimal resource-management implementation 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 e.g., 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 autonomically switched for 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, etc. However, there is a tradeoff 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 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 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 all 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 cover-

age. 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 technique whereby insertion of test probes across the chip allows intermediate values to be examined.

3 Autonomically improving performance

Autonomic computing covers a wide range of goals; for systems software one of the most important is to be able to self-tune to maintain or improve performance. In this section, we discuss how hot-swapping can support new, and extend existing performance enhancements, allowing the operating system (OS) to tailor itself to a changing environment.

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

An example of this is file sharing. While 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% or more [1]. 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.¹

¹Other examples where similar optimizations are possible are 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 network connections that have a single client on the system (in which case data can be shared with zero copy between the

Optimizing for file attributes: Several specialized file system structures have been proposed to optimize file layout and caching for files with different attributes [2, 3]. Dynamic replacement can take advantage of these different structures by implementing each, and switching between them when appropriate.

For example, while the vast majority of files accessed are small ($< 4\text{KB}$), OSes must support both large files and files that grow. Dynamic replacement can take advantage of file size to optimize application performance. In K42, in the case of a small unshared file, 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.

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 [4, 5]. Researchers have shown up to 30% 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 under-utilized in today's multi-platform OSes. To ensure portable code, without making global code paths unduly complex, these features are generally either crippled or ignored entirely because implementers need to provide a single implementation to be used across all platforms. For example, there is only limited support today for large pages even though a large number of processors support them. Hot swapping makes it easier to take advantage of architectural features because special purpose objects can be introduced and used without requiring that all corner case functionality be implemented in

network service and the client).

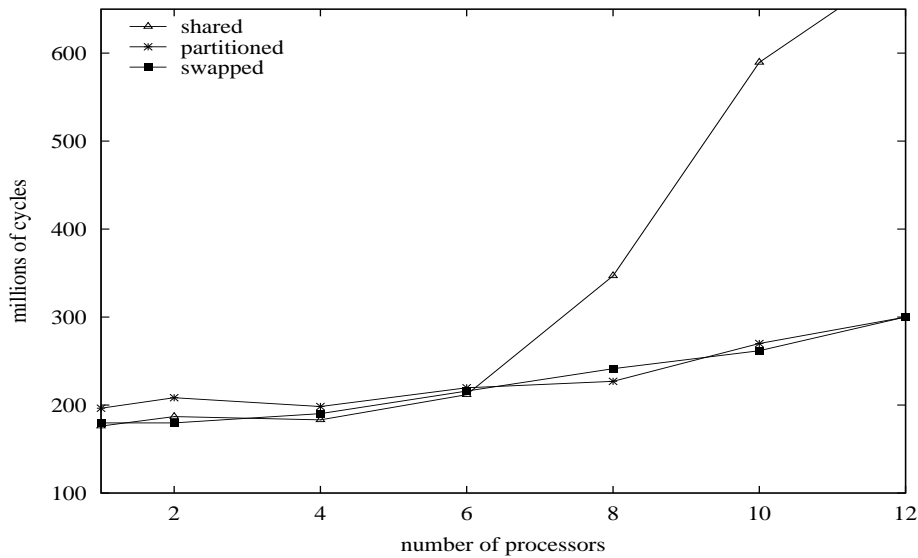


Figure 1: Concurrent searching of a 52 MB file (low is good)

every object.

Multiprocessor optimizations: In large multiprocessor systems, parallel applications can benefit from processor locality. To exploit this locality, some OSES implement services in a partitioned fashion. However, these partitioned implementations consume more memory and incur larger overheads on some operations, e.g., file destruction and process destruction. Conversely, shared implementations can minimize space and time overheads for sequential applications.

Figure 1 illustrates the performance advantages in a file-searching application 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 is cycles; lower is better. The figure shows that the shared implementation has a 10% 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% worse with twelve applications on twelve 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 OSes 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 [6, 7, 8, 9]. For example, the Exokernel's Cheetah web server demonstrated factor of two-to-four increases in throughput from network stack and file cache specializations [10]. 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 [11] and storage latency estimation descriptors [12] have shown over 100% performance increases for applications, but require detailed knowledge about the state of system structures. Inserting the necessary profiling information can affect the performance of applications that do not require this information. Hot swapping allows applications to gather more information about the state of system structures by interposing monitoring objects into the kernel. 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 researchers ability to consider algorithms designed for rare conditions that may be important for certain applications.

Supporting specialized workloads: With existing monolithic systems, optimizations introduced for one workload often negatively impact the performance of other workloads. One strategy being considered by some groups is to ship multiple versions of an OS, where each version is tuned for a particular critical workload. Another approach is through incremental specialization [13], where specific portions of the kernel are recompiled to optimize them for a particular workload. Using hot-swapping, the system can dynamically replace these optimized components to handle each specific workload. Using incremental specialization, Pu et.al. reported performance improvements as high as 70% for small amounts of data read from files.

4 Hot-Swapping infrastructure

Achieving a generic hot-swapping mechanism in system software requires a careful design. In addition to the impact on the surrounding system infrastructure, there are a set of actions involved in performing a hot-swap, including triggering, choosing the target, swapping, transferring state, and potentially adding object types. In this section, we first describe system requirements for supporting hot-swapping, both interposition and replacement, and then describe the steps involved in performing an object switch. We finish by comparing hot-swapping to adaptive code.

4.1 System structure

Many large systems, like 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. Well-defined components are necessary in order to be able to replace them with different implementations. Any code, whether it is the kernel, a database, a web server,

or any other server or application at user-level can use the hot-swapping infrastructure. The code desiring to perform the hot swap need only be structured so that there are identifiable components that can be 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 application needs because a given 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 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 OSes are not designed in a fashion that allows for hot swapping. However, the Unix Vnode 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 OSes 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 it's well-defined VFS interface. As systems become more complex, and autonomic computing becomes more important, the pressures on systems to adopt such designs will increase.

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 is appropriate to non-object-oriented systems that support hot-swapping.

4.2 Performing hot-swapping

Perhaps surprisingly, only a small number of research groups have looked into hot-swapping [14, 15, 13], and even then, their approaches have been limited to restrictive conditions. One of the reasons may be the difficulty in providing a general and efficient service that can safely and efficiently handle switching and inserting components on a multiprocessor in a multi-threaded and preemptive environment. A brief example demonstrating the difficulties is replacing the TCP/IP stack. To do so requires: *(i)* synchronizing the switching with a potentially large number of concurrent calls from applications and various parts of the operating system; *(ii)* atomically transferring state that may include active connections, buffers, and outstanding timers; *(iii)* installing the new object in the system so that its clients automatically and transparently use the new implementation.

The complexity of hot swapping live 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 describe how we have implemented the infrastructure in K42.

4.2.1 Triggering hot-swapping

In many cases we expect that objects will trigger a replacement themselves. For example, if an object is designed to support small files, but notices an increase in size, the object can trigger a swap to an implementation that supports large files. In some cases, we expect the system infrastructure to determine the need for a dynamic object replacement. Monitoring will be 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 to

identify excessive paging, and then start interposing additional monitoring on objects that cache files 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 like 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 to back a specific region of memory.

In the future, we expect that developers of autonomic computing systems will provide databases that identify changes, e.g., bug fixes, security upgrades, etc, such as in Linux distributions including RedHat 7.3. This will allow systems to periodically query these databases, download new components, and swap them in without disrupting running applications.

4.2.2 Choosing the target

In some cases, the initiator of a swap can identify the target directly, for example, in 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. While 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.

4.2.3 Doing the swap

In our experience, the majority of complexity involves performing the swap, including getting the object in a state suitable for swapping and performing the swap in a scalable manner. 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 Section 5 we discuss the infrastructure for a mechanism in K42 that provides a generic hot-swap mechanism.

4.2.4 Transferring state

A key issue is how to transfer state efficiently between the source and 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.

4.2.5 Dynamically adding object types

Downloading new code into the operating system 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

| 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: **Comparison of how autonomic features are realized with adaptive code and hot-swapping.** Both adaptive code and hot-swapping achieve some aspects of autonomic computing. Hot-swapping does so with less complexity, less normal-mode overhead, and less programmer omniscience, and enables useful features such as on-line patches.

that class in order to be able to replace it. Second, if library or subsystem code is changed, it is necessary to download code into all running applications and subsystems using that library.

4.3 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 handle varying environments is to use adaptive code. While adaptive code may not achieve the full autonomic vision outlined in Section 2, 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 choice of implementation to use can vary over time [16, 4, 17]. Adaptive code is a combination of several individual algorithms, each designed for a particular workload.

Table 1 and Figure 2 compare adaptive code to hot swapping. When used for optimizing performance, the difficulties with adaptive code can be classified in three categories: foreknowledge requirements, increase in code complexity, and performance overhead.

1. Adaptive code allows the system to switch between a set of pre-programmed algorithms.

The set of available algorithms, the monitoring code used to choose between them, and the decision-making code itself cannot be changed once the system is running.

2. Adaptive code designed for different situations 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 entire system is more complicated than allowing each component to make local adaptation decisions.
3. Adaptive code tends to lack interposition capabilities, and thus imposes some monitoring overhead on all requests. It is challenging to achieve the right level of monitoring infrastructure to understand both periods of stable and highly loaded system behavior.

A more fundamental limitation with adaptive code is that it cannot be used 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 offline and restart the system, but this incurs downtime and disrupts 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 re-implement pieces of the complexity in each service performing adaptation.

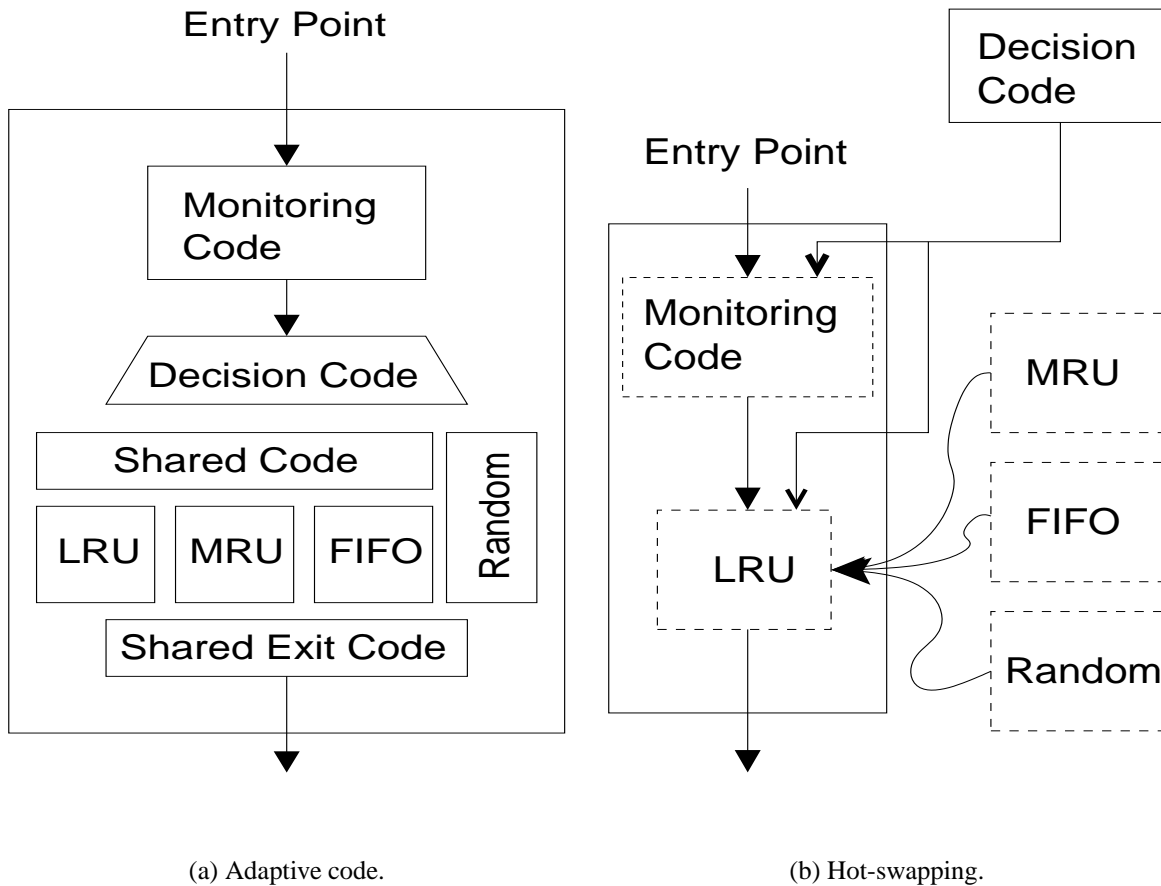


Figure 2: Adaptive Code vs. Hot-Swapping. This figure illustrates the same function implemented in two styles. The adaptive code approach is monolithic and includes monitoring code that allows the adaptive algorithm to choose a particular code path. Algorithm options must be part of the original code, 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 swapped in when desired. 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 (random doesn’t need) needs to be integrated into the code paths for the adaptive case and is inherited into each object in the hot-swapping case.

5 Hot swapping in K42

In this section, we describe K42's generic hot-swapping mechanism. To provide context, we start by presenting the overall structure and design of K42. We then describe K42's infrastructure for hot-swapping, give its status and present an example performance study. We finish with continuing work.

5.1 K42

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

5.1.1 K42 motivation and goals

K42 focuses on achieving good performance and scalability, providing a customizable and maintainable system, and being accessible to a large community through an open source development model. To that end, K42 fully supports the Linux API and ABI 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 SDET to significant subsystems like Apache. 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.

Providing a well-structured kernel is a primary goal of the K42 project, but performance is also a central concern. Some research operating system projects have taken particular philosophies and have followed them rigorously to extremes in order to fully examine their implications. While we

follow a set of design philosophies in K42, we're 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 and global data structures and locks, 5) moving system functionality to application libraries, and 6) moving system functionality from the kernel to server processes.

Goals of the K42 project include:

- **Performance:** A) Scale up to run well on large multiprocessors and support large-scale applications efficiently. B) Scale down to run well on small multiprocessors. C) Support small-scale applications as efficiently on large multiprocessors as on small multiprocessors.
- **Customizability:** A) Allow applications to determine (by choosing from existing components or by writing new ones) how the operating system manages their resources. B) Automatically have the system adapt to changing workload characteristics.
- **Applicability** A) Effectively support a wide variety of systems and problem domains. B) Make it easy to modify the operating system to support new processor and system architectures. C) Support systems ranging from embedded processors to high-end enterprise servers.
- **Wide availability:** A) Be available to a large open-source and research community. B) Make it easy to add specialized components for experimenting with policies and implementation strategies. C) Open up for experimentation parts of the system that are traditionally accessible only to experts.

5.1.2 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 infrastructure, base scheduling, networking, device support, etc. (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 file server, name server, socket server, pty 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 inter-process communication (IPC) 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 will support the Linux 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 system calls and implementing them with

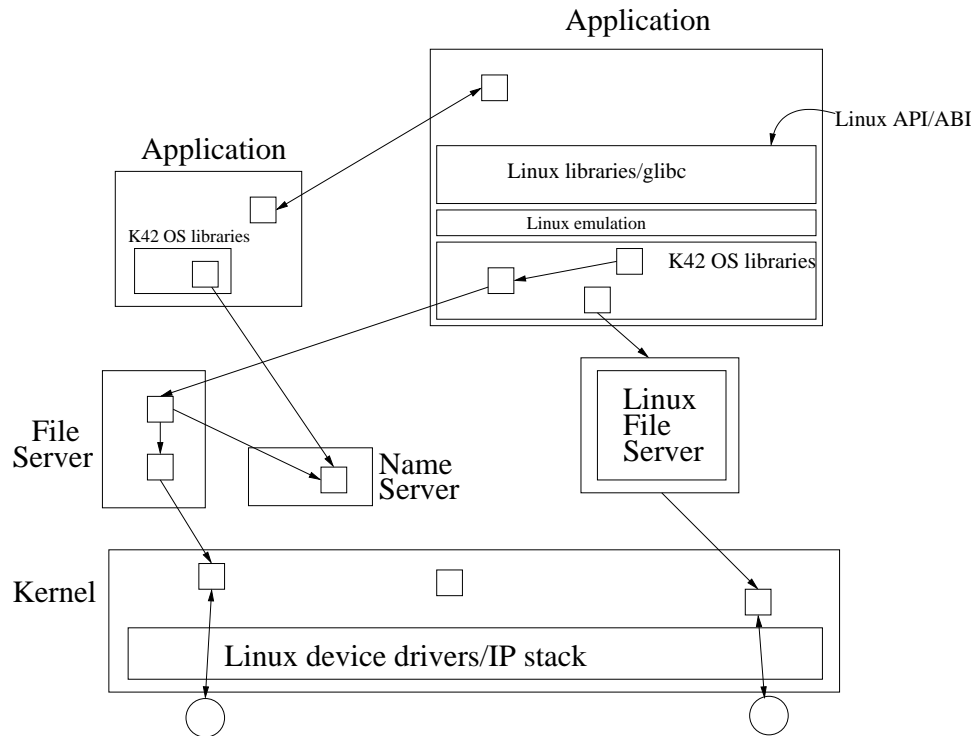


Figure 3: Structural Overview of K42

K42 code. While 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.

5.1.3 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) describing these technologies in greater de-

tail; 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. The technologies used by the infrastructure are discussed in **K42 features used** of Section 5.2.

- Object-oriented technology has been applied to the entire system. This has been used to: achieve good performance through customization, achieve good MP performance by increasing locality, increase maintainability by isolating modifications, 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.
- Much system functionality has been implemented in user-level servers with good performance maintained via efficient IPCs similar to L4 [18].
- K42 uses *processor-specific* memory (the same virtual address on different processors maps to different physical addresses) to achieve good scalable NUMA 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* [19] 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 is 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 [19] similar to RCU [20] allowing objects to release their locks before calling other objects. This efficient programming model is crucial for multiprocessor performance and is similar to type-safe memory [21].

5.1.4 K42 overall status

K42 is available under an LGPL license. A tar file of the source is available at <http://www.research.ibm.com/K42>.

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, and we expect that policies and implementations studied in this framework will be transferred into vanilla Linux. Also, in the long term, we expect that the kind of technologies we are exploring with K42 will be important to Linux.

K42 currently runs on 64-bit MIPS (NUMA) and PowerPC (SMP) 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 operating systems. 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, however, still edge conditions that have not yet been addressed, and there are still objects with only simplistic implementations.

5.2 Infrastructure for hot-swapping

In K42, each virtual resource instance, e.g., a particular file, open file instance, memory region, is implemented by combining a set of (C++) object instances we call building blocks [22]. 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 The outline of our algorithm is given here and described in more detail below: *(i)* instantiate the replacement component, *(ii)* establish a quiescent state for the component to be replaced, *(iii)* transfer state from the old component to the new component,

(*iv*) swap the new component replacing all references, and (*v*) deallocate the old component.

There are three key issues that need to be addressed in this design: The first, is how to establish a quiescent state so that it is safe to transfer state and swap references. Here we describe some possible alternatives and their difficulties, and then next we present the algorithms used in K42. 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.

The second issue is deciding 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.

The final key issue is how 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 Java, 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++ [22], 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 non-object-oriented system that uses operations tables, such as vnodes.

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* [19]. Another method for doing this would be the dynamic linking technology such as that used in ELF, 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.² This mechanism is used to achieve a quiescent state for an object. The mechanism exploits the fact that operating systems are event-driven, where most requests are serviced and complete 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.

²A similar mechanism has been used by Sequent's NUMA-Q for the same reason we originally developed it for K42, namely to improve multiprocessor performance by deferring expensive, but non-critical operations [23].

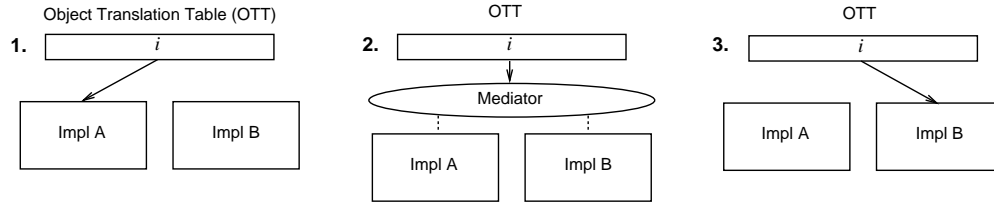


Figure 4: Steps in switching from implementations A to B of object i .

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, it contains a pointer to the old object. In the second stage, a mediator is interposed. The third stage is when the OTT is in its final state, with a pointer 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 non-erroneous 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 *Block* 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 canonical 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 *Block* 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 over-

head 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 multi-threaded multi-processor 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.

5.3 Status, performance, and continuing work

K42 fully supports the hot-swapping infrastructure described in the previous section. We have used it from simple applications, such as the search program presented in Section 3, to complex applications such as the the Spec SDET benchmark. The code works for all the objects in our system, and the main limitation currently is the number of choices we can swap between because the project's focus has not been to take time to provide many alternative implementations. We are continuing to explore using hot swapping to fulfill more of the goals of autonomic computing. Next we present an example performance study of a significant application and then finish by describing our continuing work.

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 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 commands³. 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% 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% 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.

³Due to tool chain issue we used a modified version that does not include a compiler or ps.

The above experiment shows the power of being able to use hot swapping. In other work we have examined the performance advantages of hot swapping. These results have indicated the potential of our approach, but there is still much to do within K42 to get hot swapping to a mature state. Currently the trigger for an object switch is either specified by an object or via an explicit request by an application. While 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 infrastructure and provide a more generic mechanism for gathering this information.

The decision of the target object is today made 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 autonomously swap in a security patch, or achieve higher availability by live upgrade. We are beginning to explore other autonomic advantages.

There is still tremendous potential and work 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 to continue to move forward with this research.

6 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 [15], 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)* [24] is a middleware application that offers an abstraction layer between applications and the underlying operating system. *vOS* allows modules to be refreshed, i.e., to be dynamically reloaded to achieve software rejuvenation. It does not address loading of new implementations, and its approach does not apply to operating system components.

Pu et al. describe a “replugging mechanism” for incremental and optimistic specialization [13], 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 non-scalable. 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 [14]. 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 [23], 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 operating systems more adaptive and extensible as in SPIN [6], Exokernel [7], and VINO [9]. 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, multi-threaded MP system. Several people have also done work on adding extensibility to both applications and systems. CORBA [25], DCE [26], and RMI [27] 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.

7 Conclusions

Autonomic system software must possess many capabilities to maintain themselves and to run applications well on different hardware platforms and across different environments. We proposed an object-oriented system structure that supports hot swapping as an infrastructure to meet these autonomic challenges by dynamically monitoring and modifying the operating system. We described how hot swapping can yield performance advantages and showed an example from K42. We demonstrated the technical feasibility of the hot-swapping approach by implementing it in K42 and presented our implementation. We have used K42's hot-swapping infrastructure to achieve better performance and a more maintainable system. We continue to explore the autonomic vision in K42 examining security, upgrade, and multiple object coordination issues.

Our system is available open-source. Please see our home page (<http://www.research.ibm.com/-K42/>) if you would like to participate in this research project and for additional white papers on K42.

8 Acknowledgments

We would like to thank for the referees for their positive and helpful comments on our paper.

References

- [1] O. Krieger, M. Stumm, and R. Unrau. The Alloc Stream Facility: a redesign of application-level stream I/O. *IEEE Computer*, **27**(3):75–82, March 1994.
- [2] O. Krieger and M. Stumm. HFS: a performance-oriented flexible file system based on building-block compositions. *ACM Transactions on Computer Systems*, **15**(3):286–321. ACM, 1997.
- [3] ReiserFS home page, April 2002. <http://www.namesys.com>.
- [4] 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, Washington, United States), pages 115–126. ACM Press, 1997.
- [5] 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. *Symposium on Operating Systems Design and Implementation* (San Diego, CA), pages 119–134. USENIX Association, 23–25 October 2000.
- [6] B. N. Bershad, S. Savage, P. Pardy, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, **29**(5), 3–6 December 1995.
- [7] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an operating system architecture for application-level resource management. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, **29**(5), 3–6 December 1995.
- [8] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. *Symposium on Operating Systems Design and Implementation* (Seattle, WA). Published as *Operating Systems Review*, **30**:153–167. ACM, 1996.
- [9] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. *An introduction to the architecture of the VINO kernel*. TR-34–94. Harvard University, 1994.
- [10] 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. ACM, February 2002.
- [11] 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**(2):111–170. ACM, 2001.
- [12] R. V. Meter and M. Gao. Latency management in storage systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA), pages 103–117. USENIX Association, 23–25 October 2000.
- [13] 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. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO). Published as *Operating Systems Review*, **29**(5), 3–6 December 1995.
- [14] 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, Utah, United States), pages 13–23. ACM Press, 2001.
- [15] G. Hjalmytsson and R. Gray. Dynamic C++ classes: a lightweight mechanism to update code in a running program. *Annual USENIX Technical Conference*, pages 65–76. USENIX Association, June 1998.

- [16] 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**(2):242–283, 1997.
- [17] 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.
- [18] J. Liedtke. On micro-kernel construction. *Proceedings of the fifteenth ACM symposium on Operating systems principles* (Copper Mountain, Colorado, United States), pages 237–250. ACM Press, 1995.
- [19] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA), pages 87–100, 22–25 February 1999.
- [20] 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), 26–29 June 2002.
- [21] 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, Washington, United States), pages 123–136. ACM Press, 1996.
- [22] M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm. Customization lite. *Hot Topics in Operating Systems*, pages 43–48. IEEE, May 1997.
- [23] 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.
- [24] T. Boyd and P. Dasgupta. Preemptive Module Replacement Using the virtualizaing Operating System. *Workshop on Self-Healing, Adaptive and self-managed Systems (SHAMAN)*, June 2002.
- [25] Z. Yang and K. Duddy. CORBA: a platform for distributed object computing. *Operating Systems Review*, **30**(2):4–31, April 1996.
- [26] *Distributed Computing Environment: overview*. OSF-DCE-PD-590-1. Open Software Foundation, May 1990.
- [27] Java Remote Method Invocation - Distributed Computing for Java, November 1999. <http://java.sun.com/marketing/collateral/javarmi.html>.