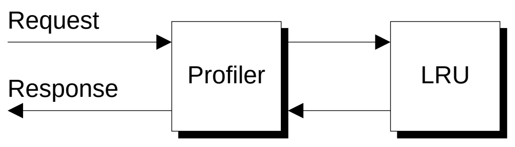
# System Support for Online Reconfiguration

Craig A. N. Soules [†] Jonathan Appavoo [‡] Kevin Hui [‡] Dilma Da Silva [§]
Gregory R. Ganger [†] Orran Krieger [§] Michael Stumm [‡] Robert W. Wisniewski [§]
Marc Auslander [§] Michal Ostrowski [§] Bryan Rosenburg [§] Jimi Xenidis [§]

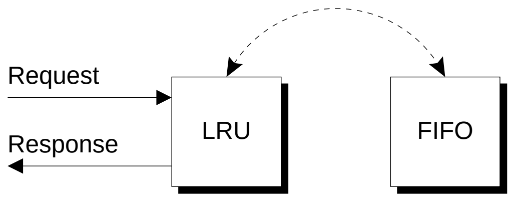
## Abstract

Online reconfiguration provides a way to extend and replace active operating system components. This provides administrators, developers, applications, and the system itself with a way to update code, adapt to changing workloads, pinpoint performance problems, and perform a variety of other tasks all while the system is running. With generic support for interposition and hot-swapping, a system allows active components to be wrapped with additional functionality or replaced with a different implementations that have the same interfaces.

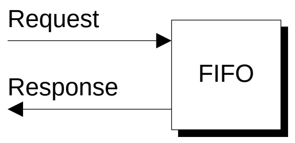
This paper identifies four support mechanisms required by online reconfiguration. It describes each mechanism's implementation in the K42 operating system, and how they are combined to implement interposition and hot-swapping. It then illustrates some dynamic performance enhancements achievable with K42's online reconfiguration including: adaptive algorithms, common case optimizations, and workload specific specializations.



(a) After profiler is interposed around the page manager.



(b) LRU page manager before hot-swap with FIFO.



(c) After FIFO page manager is fully swapped.

Figure 1: **Online reconfiguration.** This figure shows two online reconfiguration mechanisms: interposition and hot-swapping. (a) shows an LRU page manager and an interposed profiler that can watch the component's calls/returns to see how it is performing. If it determines that performance is poor, it may decide to switch the existing component for another. (b) shows the LRU page manager as it is about to be swapped with a FIFO page manager. Once the swap is complete, as shown in (c), the LRU page manager can be deleted.

## 1 Introduction

Operating systems are big and complex. They are expected to serve many needs and work well under many workloads. They are meant to be portable and work well with varied hardware resources. As one would expect, this demanding set of requirements is difficult to satisfy. As a result, patches, updates, and enhancements are common. In addition, tuning activities (whether automated or human-driven) often involve dynamically adding monitoring capabilities and then reconfiguring the system to better match the specific environment.

Common to all of these activities is a need to modify system software after it has been deployed. In some cases, an approach of shutting down the system, updating the software, and restarting is sufficient. But, this approach comes with a cost in system availability and (often) human administrative time. Also, restarting the system to add monitoring generally clears the state of the system. This can render the new monitoring code ineffective unless the old state of the system can be reproduced.

Online reconfiguration can provide a useful foundation for enhancement of deployed operating systems. With generic support built into an OS's core, the activities listed above could be more easily applied and more widely used without adding new complexity to individual subsystems. For example, patches and updates could be applied to the running system, avoiding down-time and associated human involvement. In addition, monitoring code could be dynamically added and removed, gathering system measurements with less common-case overhead.

[†] Carnegie Mellon University
[‡] University of Toronto
[§] IBM T. J. Watson Research Center

This paper describes generic online reconfiguration support and how it has been integrated into the core of the K42 operating system [27]. Figure 1 illustrates two basic mechanisms for online reconfiguration: interposition and hot-swapping. Interposition wraps an active component, extending its functionality with wrapper code that executes before and after each call to the component. Hot-swapping replaces an active component with a new implementation. Both modify an active component while maintaining availability of the component's functionality. Combined, these two mechanisms provide substantial capability for online reconfiguration.

Interposition and hot-swapping require four capabilities. First, the system must be able to identify and encapsulate the code and data for a swappable component. Second, the system must be able to quiesce a swappable component such that no external references are actively using its code and data. Third, the system must be able to transfer the internal state of a swappable component to a replacement version of that component. Fourth, the system must be able to modify all external references (both data and code pointers) to a swappable component. These four capabilities enable both mechanisms. Hot-swapping consists of the four capabilities in sequence. Interposition relies mainly on the fourth capability to redirect external references to the interposed code.

Our implementations of hot-swapping and interposition are in the context of the K42 operating system. K42's object-oriented design provides a clear swappable component choice: the object instance. The four requirements for interposition and hot-swapping in K42 are handled as follows. First, object instances are identifiable and self-contained by design. Second, a thread generation count and selective thread blocking are used to achieve and identify a quiescent state for an object instance. Third, a mechanism for data format negotiation helps designers of component implementations to reduce the cost of state transfer. Fourth, object references are routed through a level of indirection, which can be updated to modify external references.

These support mechanisms also contribute to K42 in other ways. For example, an object-oriented design leads to cleaner system structure and more scalable multiprocessor performance. Also the thread generation count is used to eliminate most component existence locks. At the same time, none of these mechanisms are specific to K42; many existing systems have one or more of these mechanisms in place, and they could add the remaining functionality to support online reconfiguration.

To illustrate the value of online reconfiguration, we explore a number of examples of its use for dynamic performance tuning within K42. For example, online reconfiguration allows clean common-case optimization; a 17% improvement is measured for dynamically switching from a non-shared implementation to a shared implementation of file caching only when sharing is detected. Similarly, swapping to specialized caching policies for small-file access gives up to a 57% performance improvement. Online reconfiguration also allows dynamic selection from several object implementations based on workload. For example, a replicated page manager gives up to a 92% improvement on some workloads, but it suffers a 7% performance loss on others. Switching between replicated and non-replicated implementations allows the system to perform well in both cases.

The rest of this paper is structured as follows. Section 2 motivates our focus on online reconfiguration. Section 3 discusses the system support required for online reconfiguration. Section 4 describes implementations of interposition and hot-swapping in the K42 operating system. Section 5 analyses the performance costs of hot-swapping and interposition and examples of their use for dynamic performance tuning. Section 6 discusses how online reconfiguration can be applied to other systems and open issues of a generic scheme for online reconfiguration. Section 7 discusses related work.

## 2 Why Online Reconfiguration

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

This section makes a case for integrating into system software a generic infrastructure for extending and replacing active system components. First, it discusses two aspects of online reconfiguration: interposition and hot-swapping. Then, it discusses a number of common OS improvements, and how interposition and hot-swapping can simplify and enhance their implementation.

### 2.1 Online reconfiguration

Once in place, online reconfiguration support can simplify the dynamic updates and changes wanted for many classes of system enhancement. Thus, a generic infrastructure can avoid a collection of similar reconfiguration mechanisms. Two mechanisms that provide such a generic infrastructure are interposition and hot-swapping.

*Interposition* wraps an active component's interface, extending its functionality. Interposition wrappers may be specific to a particular component or generic enough to wrap any component. For example, a generic wrapper might measure the average time threads spend in a given component. A component-specific wrapper for the fault handler might count page faults and determine when some threshold of sequential page faults has been reached.

*Hot-swapping* replaces an active component with a new component instance that provides the same interface and functionality. To maintain availability and correctness of the service provided, the new component picks up where the old one left off. Any internal state from the old component is transfered to the new, and any external references are relinked. Thus, hot-swapping allows component replacement without disrupting the rest of the system.

## 2.2 Applying online reconfiguration

Interposition and hot-swapping are general tools that can provide a foundation for dynamic OS improvement. The remainder of this section discusses how some common OS enhancements map onto them.

**Patches and updates**: As security holes, bugs, and performance anomalies are identified and fixed, deployed systems must be repaired. With hot-swapping, a patch can be applied to a system immediately without the need for down-time (scheduled or otherwise). This capability avoids a trade-off among availability and correctness, security, and better performance.

**Optimizing the common case**: For many OS resources, the common access pattern is simple and can be implemented efficiently. However, supporting all of the complex, uncommon cases often makes the implementation expensive. To handle these cases, a system with online reconfiguration can hot-swap between a component specialized for the common case and the standard component that handles all cases. Another way of getting this behavior is with an IF statement at the top of a component with both implementations. A hot-swapping approach separates the two implementations, simplifying testing by reducing internal states and increasing performance by reducing negative cache effects of the uncommon case code [35]. Section 5.3.2 evaluates the use of online reconfiguration to optimize exclusive access to a file, while still supporting full sharing semantics when necessary.

**Adaptive algorithms**: For many OS resources, different algorithms perform better or worse under different conditions. Adaptive algorithms are designed to combine the best attributes of different algorithms by monitoring when a particular algorithm would be best and using the correct algorithm at the correct time. Using online reconfiguration, developers can create adaptive algorithms in a modular fashion, using several separate components. Each independent algorithm can be developed as a separate component, hot-swapped in when appropriate. Also, interposed code can perform the monitoring, allowing easy upgrades and paying performance penalties only during sampling. Section 5.3.4 evaluates using online reconfiguration to provide adaptive page replacement.

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

**Application-specific optimizations**: Application specializations are a well known way of improving a particular application's performance based on knowledge only held by the application [15, 17, 19, 46]. Using online reconfiguration, an application can provide a new specialized component and swap it with the existing component implementation. This allows applications to optimize any component in the system without requiring system developers to add explicit hooks to replace each one.

**Third-party modules**: An increasingly common form of online reconfiguration is loadable kernel modules. Particularly with open-source OSes, such as Linux, it is common to download modules from the web to provide functionality for specialized hardware components. In the case of Linux, the module concept also has a business benefit, because a dynamically loaded module is not affected by the GNU Public License. As businesses produce value-adding kernel modules (such as "hardened" security modules [24, 41]), the Linux module interface may evolve from its initial focus on supporting device drivers toward providing a general API for hot-swapping of code in Linux. The mechanisms described in this paper are a natural endpoint of this evolution, and the transition has begun; we have worked with Linux developers to implement a kernel module removal scheme using quiescence [32].

### 2.3 Summary

Online reconfiguration is a powerful tool that can provide a number of useful benefits to developers, administrators, applications, and the system itself. Each individual example can be implemented in other ways. However, generic support for interposition and hot-swapping can support them all with a single infrastructure. By integrating this infrastructure into the core of an OS, one makes it much more amenable to subsequent change.

## 3 How Online Reconfiguration

Online reconfiguration has four requirements. First, components must have well-defined boundaries (i.e., a component should have well-defined interfaces that encapsulate its functionality and data). Second, it must be possible to force an active component into a quiescent state long enough to complete state transfer. Third, there must be a way to transfer the state of an existing component to a new component instance. Fourth, it must be possible to update external references to a component.

### 3.1 Component boundaries

Each system component must be self-contained with a well-defined interface and functionality. Without clear component boundaries, it is not possible to be sure that a component is completely interposed or swapped. For example, an interposed wrapper that counts active calls within a component would not notice calls to unknown interfaces. Similarly, any component that stores its state externally cannot be safely swapped, since any untransferred external data would likely lead to improper or unpredictable behavior.

Achieving clear component boundaries requires some programming discipline and code modularity. Using an object-oriented language can help. Components can be implemented as objects, encapsulating functionality and data behind a well-defined interface. Object boundaries help prevent confusing code and data sharing, often resulting in cleaner components and a more maintainable code base. However, regardless of these benefits, any solution relies on developer diligence and a strict adherence to the programming discipline.

### 3.2 Quiescence

Before a component can be swapped, the system must ensure that all active use of the state of the component has concluded. Without such quiescence, active calls could change state while it is being transfered, causing unpredictable behavior. Quiescence can be achieved by blocking incoming calls to the component and waiting for active calls to complete. Once no more calls are active within the old component, state can be safely transfered to the new component. Blocked calls can then be unblocked and processed by the new component.

One way to achieve quiescence is through the use of a reader-writer lock around the component. Each entering call grabs a read-lock on the component. To achieve quiescence, the system asks for a write lock on the component. This blocks all new incoming calls (since they require the lock to proceed), and also waits until all existing calls complete (since the write-lock is exclusive).

This naïve approach has three drawbacks. First, the system must pay an overhead on each component call to acquire and later release the component lock. Second, placing a lock around each component makes the component implementation more difficult; mishandling lock releases could easily lead to a deadlock situation. Third, the lock implementation must be able to support recursive calls, since a recursive caller inside a component pending delete could create deadlock. To deal with these problems, we propose a thread generation count, described in Section 4.

### 3.3 State transfer

State transfer synchronizes the state of a new component instance with that of an existing component. To complete a successful state transfer, all of the information required for proper component functionality must be packaged, transfered, and unpackaged. This requires that the old and new component agree upon both a package format and a transfer mechanism.

There is not likely to be a single "catch-all" solution; both the data set and data usage can vary from component to component. Instead, there is likely to be a variety of packaging and transfer mechanisms suited to each component. While it is impossible to predict what all of these mechanisms will be, there are likely to be a few common ones. For example, an upgraded component is likely to understand the existing component in detail. Transferring a reference to the old component should be sufficient for the new component to extract the necessary state.

Given that no single mechanism exists for state transfer, the system can still provide two support mechanisms to simplify its implementation. First, the hot-swapping mechanism should provide a negotiation protocol that helps components decide upon the most efficient transfer mechanisms understood by both. Second, components that share a common interface and functionality should understand (at the least) a single, canonical data format.

By ensuring this, component developers need only implement two state transfer functions to have a working implementation: to and from the canonical form.

### 3.4 External references

Whenever a component is interposed or hot-swapped, its external interface is handled by a new piece of code (the wrapper for interposition, the new component for hot-swapping). Because all calls must be routed through this new code, all external references to the original component must be updated.

Indirection and reference tracking are two common ways to handle this. With all references pointing to an indirection pointer, the system does not need to know about each individual reference. To update all external references, the system only needs to update the single indirection pointer. This is space and performance efficient, with small constant overhead per component. Reference counting, as is used in garbage collection [5, 11, 22], can be much more expensive, growing linearly with the number of component references. Whenever a new reference to a component is created, it is tracked by the system. If a component interface changes, then all tracked references can be found and updated.

### 3.5 Other issues

The focus of this work is on the mechanics of online reconfiguration. There are, of course, issues of safety and security involved with deciding which reconfigurations to allow [36] and containing suspect extensions [43, 45]. Other researchers and practitioners have provided a number of viable solutions to these problems, although many environments "address" these issues by trusting the administrator to choose dynamically added code wisely.

## 4 Implementation

This section describes how we integrated online reconfiguration into the K42 operating system. It overviews K42, describes which features we use, and details the implementations of interposition and hot-swapping.

### 4.1 K42

K42 is an open-source research OS for cache-coherent 64-bit multiprocessor systems. It uses an object-oriented design to achieve good performance, scalability, customizability, and maintainability. 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.

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 [3]. 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.

While we believe that online reconfiguration is useful in general systems, the object-oriented nature of K42 makes it a particularly good platform for exploring fine-grained hot-swapping and interposition.

### 4.2 Support mechanisms

The four requirements of online reconfiguration are addressed as follows.

**Component boundaries**: K42's building block approach naturally maps each system component onto a C++ language object, and enforces that the external interface to the object is published using the C++ virtual function table. Since all externally available calls are virtual, they are indirected through a single per-instance table. To ensure that this table is in a well-known location within each object, all components in the system are inherited from a single, base object. This enforces the required component boundaries without significant burden on developers.

**Quiescence**: To achieve a quiescent state, K42 makes use of a thread *generation count*. The generation count is used to track the lifetime of threads within the system. Every thread remembers the value of the generation count when it was created. The system keeps a count of the number of threads that are active within any generation. By increasing the generation count, a component can watch the counters for previous generations to detect when all of their threads have expired. For example, one way to achieve quiescence in an object using the generation count is to increase the generation count and then block all new threads that call that object. Once all of the threads created in previous generations have expired, the object is guaranteed to be in a quiescent state. This approach has the advantage that there is no overhead in the common case, unlike a locking solution, which would pay an overhead for each call to the component. However, it partially relies on system threads being shortlived (which is the case in K42).

**State transfer**: To assist state transfer, K42's online reconfiguration mechanism provides a *transfer negotiation protocol*. For each set of functionally compatible components, there must be a set of state transfer protocols that form the union of all possible state transfers between these components. For each component, the developers must create a prioritized list of the state transfer protocols that it supports. For example, it may be best to pass internal structures by memory reference, rather than copying the entire structure; however, both components must understand the same structure for this to be possible. Before initiating a hot-swap, K42 requests these two lists from the old and new component instances. After determining the most desirable format based on the two lists, it requests the correct package format from the old component and passes it to the new component. Once the new component has unpackaged the data, the transfer is complete.

**External references**: K42 uses its *object translation table* to provide a layer of indirection for accessing system components. When an object instance is created, an entry for it is created in the object translation table, and all external calls to the component are made through this reference. K42 can perform a hot-swap or interposition on this component by updating its entry in the table. Although this incurs an extra pointer dereference per component call, the object translation table has other benefits (e.g., improved scalability [18]) that outweigh this overhead.

## 4.3 Online adaptation

The remainder of this section describes how K42 utilizes the four system support features described above to provide interposition and hot-swapping.

### 4.3.1 Interposition

Object interposition interposes additional functionality around all function calls to an existing object instance. We partition interposed functionality into two pieces: a *generic interposer* and a *wrapper object*.

The first step by the *generic interposer* is to replace the original object in the object translation table with a pointer to itself. All subsequent calls now go through the interposer, requiring that it provide transparent call forwarding to the component behind it.

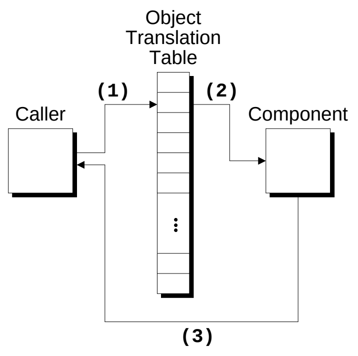
To handle arbitrary object interfaces, K42's interposer leverages the fact that every external call goes through a virtual function table. Once the generic interposer replaces the original object in the object translation table, all calls go through the interposer's virtual function table. The interposer overloads the method pointers in its virtual function table to point at a single *interposition method*, forcing all external calls through this function. The interposition method handles calls to the wrapper object's methods as well as calling the appropriate method of the original component.

To handle arbitrary call parameters and return values, the interposer method must ensure that all register and stack state is left untouched before the call is forwarded to the original component. At odds with this requirement is the need to store information normally kept on the stack (e.g., the original return address, local variables). To resolve this conflict, the interposer allocates space for any required information on the heap and keeps a pointer to it in a callee-saved register. This register's value is guaranteed to be preserved across function calls; when control is returned to the interposer, it can retrieve any saved information. The information saved in the heap space must include both the original return address and the original value of the callee-saved register being used, since it must be saved by the callee for the original caller.

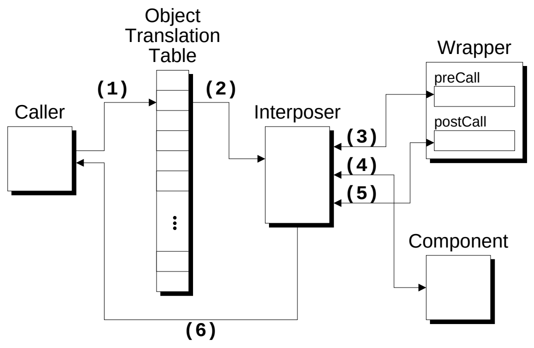
The wrapper object is a standard C++ object with two calls: PRECALL and POSTCALL. As one might suspect, the former is called before the original object's function, and the latter is called after. In these calls, a wrapper can maintain state about each function call that is in flight, collect statistical information, modify call parameters and return values, and so on.

The division of labor between the generic interposer and the wrapper object was chosen because the interposition method can not make calls to the generic interposer's virtual function table. If the wrapper and interposer were combined (using a parent interposer, and inheriting for each specific case), then the interposition method would have to locate the correct PRECALL and POSTCALL from the generic interposer's internal interfaces. Unfortunately, this is difficult, since they would be in compiler-specified locations that cannot be determined at run-time. By separating the wrapper, we can avoid specializing the interposition method for each wrapper, since the PRECALL and POSTCALL can be located using the wrapper's virtual function table.

Figure 2 shows object interposition step-by-step. Figure 2a is the initial state of the caller and component. To perform the interposition, instances of the generic interposer and the wrapper object are created. The interposer keeps a pointer to the original object and replaces its entry in the object translation table with a pointer to itself. Figure 2b shows the state of the caller and component after the object translation table is updated. At this point, all function calls to the component are now sent to the interposer. When the interposer receives an incoming call,



(a) Before interposition.



(b) After interposition.

Figure 2: **Component interposition.** This figure shows the steps of component interposition. (a) shows how callers access components through the object translation table. In this case, calls from the caller lookup the component in the object translation table, and then call the component based on that indirection. (b) shows an interposed component. In this case, the caller's indirection points the call at the generic interposer. The interposer then makes three calls, first to the wrapper's PRECALL, then the original component call, then the wrapper's POSTCALL.

it calls the wrapper's PRECALL, then calls the original object's function, then calls the wrapper's POSTCALL, and finally, returns to the original caller. Although not shown in Figure 2, the original function's call and the POSTCALL may be skipped depending upon the return value of the PRECALL.

To detach an interposed wrapper, the corresponding interposer object simply replaces its object translation table entry with a pointer to the original object. Once the object translation table is updated, all incoming calls will be sent directly to the original object. Garbage collection of the interposer and wrapper can happen out-of-band, once they quiesce.

### 4.3.2 Hot-swapping

K42's object hot-swapping mechanism builds on interposition. The first step of hot-swapping from the current object instance (X) to a new object instance (Y) is to interpose X with a *mediator*. Before the mediator can swap objects, it must ensure that there are no in-flight calls to X (i.e., the component must be in a quiescent state). To get to this state, the mediator goes through a three phase process: forward, block, and transfer.

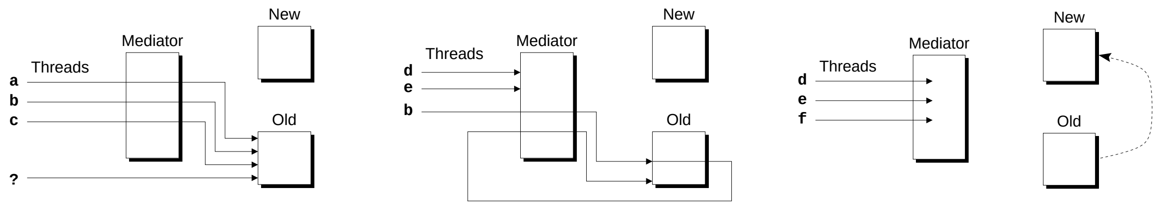
In the *forward* phase, the mediator tracks all threads making calls to the component and forwards each call on to the original component. This phase continues until it is certain that all calls started before call tracking began have completed. To detect this, K42 relies on the generation count. When the forward phase begins, a request to increase the generation count is made. Once the generation is advanced, all threads from the new generation are tracked. All other threads in the system are known to have completed once all of the earlier generation counters are zero, at which point the forward phase completes.

Figure 3a illustrates the forward phase. When the mediator is interposed, there may already be calls in progress; in this example, there is one such call marked as **?**. First the generation count is advanced, after which the new calls **a**, **b**, and **c** are tracked by the mediator. The next phase may begin once previous generations complete.

The mediator begins the *blocked* phase once all calls in the component are tracked. In this phase, the mediator temporarily blocks all new incoming calls, while it waits for the calls it has been tracking to complete. An exception must be made for incoming recursive calls, since blocking them would create deadlock. Once all the tracked calls have completed, the component is in a quiescent state, and the state transfer can begin.

Figure 3b illustrates the blocked phase. In this case, thread **b** is in progress, and must complete before the phase can complete. New calls **d** and **e** are blocked; however, because blocking **b** during its recursive call would create deadlock, it is allowed to continue.

Because the blocked phase is the only phase where threads are unable to make forward progress, it is important to make this phase as short as possible. Although a simpler implementation could remove the forward phase (blocking new calls immediately and waiting for previous generations to complete), waiting for all threads in the system to expire may take too long. Due to K42's event driven model, thread lifetimes are known to be



(a) Forward. (b) Block. (c) Transfer.

Figure 3: **Component hot-swapping.** This figure shows the three phases of hot-swapping: forward, block, and transfer. In the forward phase, new calls are tracked and forwarded while the system waits for untracked calls to complete. Although this phase must wait for all old threads in the system to complete, all threads are allowed to make forward progress. In the block phase, new calls are blocked while the system waits for the tracked calls to complete. By blocking only tracked calls into the component, this phase minimizes the blocking time. In the transfer phase, all calls to the component have been blocked, and state transfer can take place. Once the transfer is complete, the blocked threads can proceed to the new component and the old component can be garbage collected.

short. However, this may not be the case in other systems, and with the separation of the forward and block phases, the duration of the blocked phase is only dependent upon the lifetime of active threads in the component.

One tradeoff of K42's event model is that cross processor and cross address space calls are done with new threads. This means that a cyclic external call chain could result in deadlock (since the recursion would not be caught). Although developers should be careful never to create these situations, the hot-swapping mechanism prevents deadlock in these situations with a timeout and retry mechanism. If this timeout is triggered enough times, the hot-swap will return a failure.

After the component has entered a quiescent state, the mediator begins the *transfer* phase. In this phase, the mediator performs state transfer between the old and new components, updates the object translation table entry to point at the new component, and then allows blocked calls to continue to the new component. Each set of functionally compatible components share a set of up to 64 state transfer protocols. Acceptable protocols are specified using a bit vector that is returned from each component. The intersection of these vectors gives the list of potential protocols. The mediator determines the best common format by requesting the protocol vector from each component, and then choosing the protocol corresponding to the highest common bit in the vector. Once a protocol is decided upon, the mediator retrieves the packaged state from the original component and passes it to the new component.

Figure 3c illustrates the transfer phase. Once the old component is quiescent, all state is transfered to the new component. Once the unpackaging completes, threads **d**, **e**, and **f** are unblocked and sent to the new component. At this point, the mediator may be detached and the old component destroyed.

### 4.4 Multiprocessor issues

More and more, modern operating systems are interested in scaling to perform well in a multiprocessor setting. Making online reconfiguration efficient for replicated components has several complications not yet discussed. Achieving global quiescence in a multiprocessor system can take longer; it is preferable to do swapping on a per-processor basis, allowing quiescence to be reached independently on each processor. Additionally, cross-processor deadlock must be considered. A global object translation table will cause many cross processor memory references; if possible the object translation table should be replicated across processor memory. Finally, to reduce duration of blocking, state transfer should proceed in parallel. K42's implementation of online reconfiguration takes care to address these issues, and scales well to many processors, as our results will show.

### 4.5 Summary

Our solution handles call interception and mediation transparently to clients with external interfaces, and separates the complexities 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 online reconfiguration process does not require support from the component, simplifying the creation of components that wish to take advantage of interposition or hot-swapping.

## 5 Evaluation

In this section, we evaluate K42's online reconfiguration. Our evaluation comes in two parts. First, we quantify the basic overheads and latencies of interposition and hot-swapping. Second, we illustrate the use of online reconfiguration for dynamic performance enhancement with several concrete examples.

### 5.1 Experimental setup

The experiments were run on two different machines. Both of them were RS/6000 IBM PowerPC bus-based cache-coherent multiprocessors. One was an S85 Enterprise Server with 24 600MHZ RS64-IV processors and 16GB of main memory. The other was a 270 Workstation with 4 375MHZ Power3 processors and 512MB of main memory. Unless otherwise specified, all results are from the S85 Enterprise Server.

Throughout the evaluation, we use two separate benchmarks: Postmark and SDET.

**Postmark** was designed to model a combination of electronic mail, netnews, and web-based commerce transactions [28]. It creates a large number of small, randomly-sized files and performs a specified number of transactions on them. Each transaction consists of a randomly chosen pairing of file creation or deletion with file read or append. All random biases, the number of files and transactions, and the file size range can be specified via parameter settings. Unless otherwise specified, we used 1000 files, 10,000 transactions, file sizes ranging from 128B to 8KB, and even biases.

**SDET** executes one or more scripts of user commands designed to emulate a typical software-development environment (e.g., editing, compiling, and various UNIX utilities). The scripts are generated from a predetermined mix of commands,[1] and are all executed concurrently. It makes extensive use of the file system and memory-management subsystems, making it useful for scalability benchmarking. Throughout this section we will refer to an "N-way SDET" which describes running N concurrent scripts on a machine configured with N processors.

### 5.2 Basic overheads

During normal operation, the only overhead of online reconfiguration is the indirection used to update external references. In K42, this is done using the object translation table, which results in a single pointer dereference on every external component call. As we discuss in Section 6.1, K42 already pays this cost to improve multiprocessor performance. The remainder of the overheads for online reconfiguration are from the specific implementations of interposition and hot-swapping. These overheads were measured on the 270 Workstation.

| Operation | $\mu$seconds |
|---|---|
| Attach | 17.84 (0.16) |
| Component call | 1.40 (0.02) |
| Detach | 4.23 (0.49) |

Table 1: **Interposer overhead.** There are three costs of interposition: attach, call, and detach. Attaching the interposer involves initializing the interposer and wrapper and updating the object translation table. Calls to the component involve two additional method calls to the wrapper object and a heap allocation. Detaching the interposer only involves updating the object translation table, making its foreground cost zero; however, any process waiting for the detach must pay the small overhead of destroying the objects. The average cost of each operation is listed in clock ticks along with its standard deviation.

**Interposition**: There are three performance costs for interposition: attaching the wrapper, calling through the wrapper (as opposed to instrumenting the component directly), and detaching the wrapper. To measure these costs, we attached, called through, and detached an empty wrapper 100,000 times, calculating the average time for each of the three operations. Since the empty wrapper performs no operations (simply returning from the PRECALL and POSTCALL), all of the call overhead is due to interposition.

Table 1 list the costs of interposition. Attaching the interposer is the most expensive operation, involving memory allocation and object initialization; however, at no point during the attach are incoming calls blocked. Although detaching the interposer only requires updating the object translation table, the teardown of the interposer and wrapper is listed as an overhead for the process performing the detach. One simple optimization to component calls is to skip the POSTCALL whenever possible. Doing so removes the expensive memory allocation, since no state would be kept across the forwarded call (control can be returned directly to the original caller).

**Hot-swapping**: K42's *file cache manager* objects (FCMs) track in-core pages for individual files in the system. To determine the expected performance of hot-swapping, we perform a "null" hot-swap of an FCM (swapping it with itself) at points of high system contention while running a 4-way SDET. Contention is detected by many threads accessing an FCM concurrently. Although high system contention is the worst time to swap (since threads are likely to block, increasing the duration of the mediation phases), it is important to un-

[1]We do not run the compile, assembly and link phases of SDET, since, at the time of this paper, gcc executing on a 64-bit platform is unable to generate correct 64-bit PowerPC code

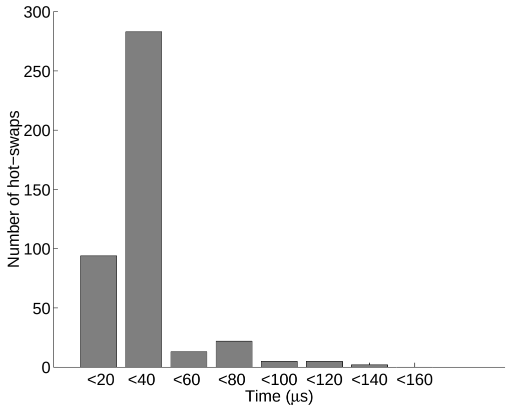


Figure 4: **Null swap.** This figure presents a histogram showing the cost of performing a null-swap at contended points in the system. For each bin, there is a count of the number of swaps with completion times that fell within that bin. On average, a swap took 27.57 $\mu$s to complete, and no swap took longer than 132.64 $\mu$s.

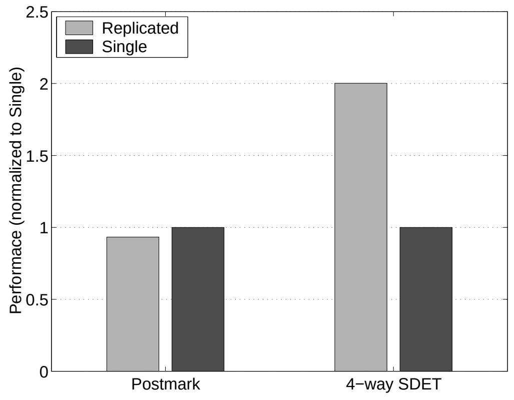


Figure 5: **Single v. Replicated.** This figure shows two different FCM implementations run under different workloads. In postmark, the single-access FCM performs better because it has less memory overhead during the creation and deletion of files. Conversely, the replicated FCM performs better under 4-way SDET because it scales well to multiple processors.

derstand this sort of "worst-case" swapping scenario.

During a single run of 4-way SDET the system detected 424 points of high contention. The average time to perform a hot-swap at these points was 27.57 $\mu$s with a 19.78 $\mu$s standard deviation, a 1.06 $\mu$s minimum and a 132.64 $\mu$s maximum. Additionally, the throughput of SDET while performing null hot-swaps and the throughput of a normal SDET run were within a standard deviation. Hot-swapping while the system is not under contention is more efficient, since the forward and block phases are shorter. Performing random null hot-swaps throughout a 4-way SDET run gave an average hot-swap time of 10.75 $\mu$s.

## 5.3 Reconfiguration for performance

This section evaluates four adaptive performance enhancements, each implemented using online reconfiguration.

### 5.3.1 Single v. Replicated

This experiment uses online reconfiguration to hot-swap between different component implementations for different workloads. We use two FCM implementations, a *single* FCM designed for uncontended use, and a *replicated* FCM designed to scale well with the number of processors. Although a single FCM uses less memory, it must pay a performance penalty for cross-processor accesses. A replicated FCM creates instances on each processor where it is accessed, but at the cost of using additional memory.

Figure 5 shows the performance of each FCM under both Postmark (using 10,000 files, 50,000 transactions and file sizes ranging from 512B to 16KB) and 24-way SDET. Because Postmark is a single application that acts on a large number of temporary files, the overhead of doing additional memory allocations for each file with a replicated FCM causes a 7% drop in performance. On the other hand, using the replicated FCM in the concurrent SDET benchmark gives performance improvements that scale from 8% on a 4-way SDET to 101% on a 24-way SDET. A replicated FCM helps SDET because each of the scripts run on separate processors, but they share part of their working set.

K42 detects when multiple threads are accessing a single file and hot-swaps between FCM implementations when appropriate. Using this approach, K42 achieves the best performance under both workloads.

### 5.3.2 Exclusive v. Shared

This experiment uses online reconfiguration to swap between an optimized non-shared component and a default shared component for correctness. We created a file handle implementation that resides entirely within the application. While this improves performance, it can only be used when an application has exclusive access to the file. Once file sharing begins, an in-server implementation must be swapped in to maintain the shared state.

Figure 6 shows the performance of swapping in the exclusive access optimization when possible in Postmark. Because most of the accesses in Postmark are exclusive, it sees a 34% performance improvement.

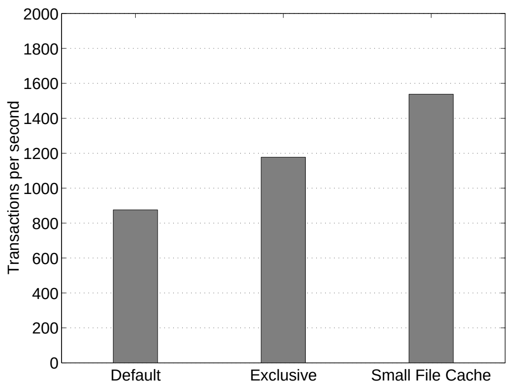


Figure 6: **Common-case optimization.** Using online reconfiguration, K42 can cache exclusive file handles within the application and swap to a shared implementation when necessary for correctness. A further enhancement is to cache small, exclusive files within the application's address space. This figure compares these three system configurations (default, exclusive, and small file cache) using Postmark. Using online reconfiguration for the exclusive case shows a 34% performance imrpovement, while the small file caching shows an additional 40% improvement beyond that.

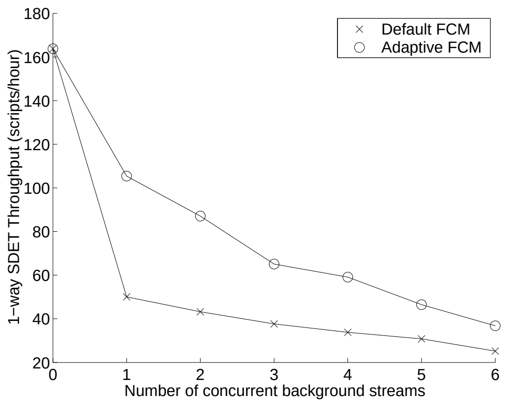


Figure 7: **Sequential page faults.** One adaptive page replacement algorithm performs MRU page replacement for sequential streams. This reduces the amount of memory wasted by streams whose pages will never be accessed again. Using online reconfiguration, K42 can detect sequential streams and swap to a sequentially optimized FCM. This figure compares the default page replacement to the adaptive algorithm by running 1-way SDET concurrently with a number of streaming applications. Using the adaptive page replacement, the system degrades more slowly, reducing the effect of streaming applications on the performance of the entire system.

### 5.3.3 Small v. Large

This experiment also uses online reconfiguration to hot-swap between a specialized non-shared component and a default shared component. In general, file data is cached with the operating system; however, access for small, exclusive files ($<$ 3 KB) can be optimized by also caching the file's data within an application's address space. While this incurs a memory overhead for double caching the file (once in the application, once in the OS), this is acceptable for small files, and it leads to improved performance.

Figure 6 shows the Postmark performance of three schemes: the default configuration, the exclusive caching scheme presented in Section 5.3.2, and application-side caching. We found that hot-swapping between caching implementations gives an additional 40% performance improvement over the exclusive access optimization.

Originally, K42 implemented this using a more traditional adaptive approach, hard-coding the decision process and both implementations into a single component. We found that reimplementing this using online reconfiguration simplified and clarified the code, and it was less time-consuming to implement and debug.

### 5.3.4 Sequential page faults

This experiment uses online reconfiguration to implement an adaptive page replacement algorithm. An interposed wrapper object watches an FCM for sequential page mappings. If the access is deemed sequential, it hot-swaps to a sequentially optimized FCM that approximates MRU page replacement. If this sequential behavior ends, the FCM is swapped back to the default FCM.

Figure 7 shows the performance of 1-way SDET in the face of competing streaming applications all run over NFS. These streaming applications access files significantly larger than the 512MB of main memory available in the 270 Workstation used in this experiment. Using the default FCM, the streaming applications quickly fill the page cache with useless pages, incurring the pager overhead immediately. This ruins SDET performance. On the other hand, using the adaptive approach, the sequential applications consume very little memory, since they throw away pages shortly after using them. This makes SDET performance degrade more slowly, meaning more streaming applications can be run while achieving the same performance as with the default FCM.

## 6 Discussion

This section is divided into two parts. First, we discuss how the support mechanisms for online reconfiguration can be applied to existing systems and, in some cases, already have. Second, we discuss open issues of a generic mechanism for online reconfiguration.

## 6.1 Application to other systems

Many of the support mechanisms used to provide online reconfiguration are useful for other reasons, and were implemented in K42 before online reconfiguration was even considered. For this same reason, many systems already contain several of the support mechanisms for online reconfiguration.

As is discussed in Section 3, an object-oriented design leads to better code structure, cleaner interfaces, more maintainable code, and a more modular approach that improves scalability. For this reason, modular approaches and object-oriented designs are becoming increasingly common in operating systems (e.g., the VFS layer in UNIX, shared libraries, plug-and-play device drivers, loadable module support). As systems incrementally add this modularity, more and more components in the system can be swapped.

K42's generation count was originally designed as a way to remove the locks used to synchronize object deletion. Instead of threads locking the object when they enter, they optimistically call the object. When an object is deleted, the generation is advanced. Once the deleted object enters a quiescent state, a garbage collector can safely delete the object. We have worked with Linux developers to add a similar mechanism for detecting quiescence, and Linux now uses it to solve the problem of safely removing kernel modules. Adding this mechanism to other systems can provide similar benefits.

K42's object translation table was originally designed as a way to improve scalability. Replicated components can improve scalability by avoiding cross-processor accesses; however, callers must know which replica to access. In K42, each processor has a local object translation table mapped into the same virtual address. Whenever a thread makes a component call through the object translation table, its call is directed to the local replica. Adding a similar level of indirection to other systems should be straightforward, and it provides benefits beyond online reconfiguration.

Although the state transfer protocol has no additional benefits beyond online reconfiguration, this mechanism is not attached to any particular design of the system. Adding this final support mechanism to any system that wished to take advantage of online reconfiguration should be straightforward.

### 6.2 Open issues

The remainder of this section discusses open issues in realizing a generic mechanism for online reconfiguration.

**Generic state transfer**: K42's hot-swapping mechanism provides a protocol for negotiating the best common format for state transfer between objects. But, it relies upon support from the components being swapped to complete state transfer. Ideally, the infrastructure would perform the entire state transfer, making hot-swapping entirely transparent. While this goal may not be fully attainable, it may be possible to provide more support than K42 currently does.

**Object creation and management**: When a performance upgrade or security patch is released for a particular component implementation, every instance of that component should be hot-swapped. Additionally, any place where an instance of the component is created must also be updated to create the new component type rather than the old one. K42's "factory" mechanism for creating and tracking objects is still under development.

**Coordinated swapping**: While hot-swapping individual components can provide several benefits, there may be times when two or more components must be swapped together. This can be achieved by having the set of mediators coordinate their phase transitions.

**Confirming component functionality**: Although K42 requires that swapped components support the same interface, it makes no guarantees that the functionality provided by the two components is the same. Although it may be possible for components to provide annotations about their functionality, the system must trust that component developers correctly support the claimed or required functionality.

**Interface management**: Currently, K42's online reconfiguration requires that swapped components support the same interface. While it is straightforward to expand these interfaces (since the old interface is a subset of the new interface), it is currently not possible to reduce interfaces; in particular, it is not possible to know if an active code path in the system relies upon a particular part of the interface. Currently, K42 relies on component developer coordination to manage component interfaces properly.

## 7 Related Work

Modifying the code of a running system is a powerful tool that has been explored in a variety of contexts. The simplest and most common example of adding new code to a running system is dynamic linking [20]. When a shared library is updated, all programs dependent on the library are automatically updated. It is also possible to update the code in running systems using shared libraries; however, the application itself must provide this support and handle all aspects of the update beyond loading the code into memory.

Hjálmtÿsson and Gray describe a mechanism for dynamic C++ objects [26]. These objects can be hot-swapped, and they do so without creating a quiescent state. To achieve this, they provide two options. First, old objects continue to exist and service requests, while all new requests go to the new object. This requires some form of coordination of state between the two co-existing objects. Second, if an object is destroyed, any active threads within the object are lost. For this reason, clients of an object must be able to detect this broken binding and retry their request.

CORBA [7], DCE [37], RMI [39], and COM [12] are all application architectures that support component replacement during program execution. However, these architectures leave the problems of quiescence and state transfer to the application, providing only the mechanism for updating client references.

Similarly, several distributed systems have examined ways to dynamically configure the location of components, requireing much of the same support [4, 31]. Bloom proposes a formal model of the criteria required to safely replace one subsystem instance with another in the context of type-safe, distributed environments [8]. Some of the analysis she performs using the Argus system may be applicable to K42's generic online reconfiguration.

Pu, et al. describe a "replugging mechanism" for incremental and optimistic specialization [38], 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 [13].

Hicks, et al. describe a method for dynamic software updating [25]. In their approach, all objects of a certain type are updated simultaneously; it is not possible to update individual instances, as is possible with our scheme. Moreover, they require that the program be coded to decide when a safe-point has been reached and initiate the update.

In addition to the work done in different reconfiguration mechanisms, many groups have applied online reconfiguration to systems and achieved a variety of benefits. Many different adaptive techniques have been implemented to improve system performance [2, 21, 29, 30]. Extensible operating systems have shown performance benefits for a number of interesting applications [6, 16, 42]. Technologies such as compiler-directed I/O prefetching [9] and storage latency estimation descriptors [34] improve application performance using detailed knowledge about the state of system structures. Incremental and optimistic specialization [38] can remove unnecessary logic for common-case accesses. K42's online reconfiguration can simplify the implementation of these improvements, removing the complicated task of instrumenting the OS with the necessary hooks to do reconfiguration on a case-by-case basis.

K42 is not the first operating system to use an object oriented design. Object-oriented designs have helped with organization [23, 40], extensibility [10], reflection [47], persistence [14], and decentralization [1, 44]. In addition, K42's method of detecting a quiescent state is not unique. Sequent is NuMAQ used a similar mechanism for detecting quiescent state [33], and recently, SuSE Linux 7.3 has integrated a mechanism for detecting quiescence in kernel modules [32].

## 8 Conclusions

Online reconfiguration provides an underlying mechanism for component extension and replacement through interposition and hot-swapping. These mechanisms can be leveraged to provide a variety of dynamic OS enhancements. This paper identifies four support mechanisms required for interposition and hot-swapping, and describes their implementation in the K42 operating system. We demonstrate the flexibility of online reconfiguration by implementing an adaptive paging algorithm, two common-case optimizations, and a workload specific specialization.

## References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. Summer USENIX Technical Conference, July 1986.

[2] 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.

[3] M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm. Customization lite. Hot Topics in Operating Systems, pages 43–48. IEEE, 1997.

[4] L. Bellissard, S. B. Atallah, F. Boyer, and M. Riveill. Distributed application configuration. International Conference on Distributed Computing Systems, pages 579–585, 1996.

[5] J. K. Bennett. *Distributed Smalltalk: inheritence and reactiveness in distributed systems.* PhD thesis, published as 87–12–04. Department of Computer Science, University of Washington, December 1987.

[6] 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. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5), 3–6 December 1995.

[7] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. International Conference on Configurable Distributed Systems, pages 35–42. IEEE Computer Society Press, 1998.

[8] T. Bloom. *Dynamic module replacement in a distributed programming system.* PhD thesis, published as MIT/LCS/TR–303. Massachusetts Institute of Technology, Cambridge, MA, March 1983.

[9] 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.

[10] R. H. Campbell and S.-M. Tan. *$\mu$Choices*: an object-oriented multimedia operating system. Hot Topics in Operating Systems, pages 90–94. IEEE Computer Society, 4–5 May 1995.

[11] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. *Modula-3 report (revised)*. 52. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 1 November 1989.

[12] Distributed Component Object Model Protocol-DCOM/1.0, http://www.microsoft.com/Com/resources/comdocs.asp.

[13] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. IFIP International Conference on Computer Security, 1996.

[14] P. Dasgupta, R. J. LeBlanc, Jr, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, **24**(11):34–44, November 1991.

[15] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: application-level virtual memory. Hot Topics in Operating Systems, pages 72–77. IEEE Computer Society, 4–5 May 1995.

[16] 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. Published as *Operating Systems Review*, **29**(5), 3–6 December 1995.

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

[18] 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, pages 87–100, February 1999.

[19] 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.

[20] R. A. Gingell. Shared libraries. *UNIX Review*, **7**(8):56–66, August 1989.

[21] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1997.

[22] J. Gosling and H. McGilton. *The Java language environment*. Technical report. October 1995.

[23] A. S. Grimshaw, W. Wulf, and T. L. Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, **40**(1):39–45. ACM Press, January 1997.

[24] Guardian Digital, Inc., http://www.guardiandigital.com/.

[25] M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. ACM, 2001.

[26] G. Hjalmtysson 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, 1998.

[27] The K42 Operating System, http://www.research.ibm.com/K42/.

[28] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[29] 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, pages 119–134. USENIX Association, 2000.

[30] 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.

[31] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. International Workshop on Configurable Distributed Systems, March 1994.

[32] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read copy update. Ottawa Linux Symposium, 2001.

[33] 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, 1998.

[34] R. V. Meter and M. Gao. Latency management in storage systems. Symposium on Operating Systems Design and Implementation, pages 103–117. USENIX Association, 2000.

[35] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of techniques to improve protocol processing latency. ACM SIGCOMM Conference. Published as *Computer Communication Review*, **26**(4):73–84. ACM, 1996.

[36] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. Symposium on Operating Systems Design and Implementation, pages 229–243. Usenix Association, Berkeley, CA, October 1996.

[37] *Distributed Computing Environment: overview*. OSF-DCE-PD-590-1. Open Software Foundation, May 1990.

[38] 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. Published as *Operating Systems Review*, **29**(5), 3–6 December 1995.

[39] Java Remote Method Invocation - Distributed Computing for Java, November 1999. http://java.sun.com/marketing/collateral/javarmi.html.

[40] V. F. Russo. *An object-oriented operating system*. PhD thesis, published as UIUCDCS–R–91–1640. Department of Computer Science, University of Illinois at Urbana-Champaign, January 1991.

[41] Security-Enhanced Linux, http://www.nsa.gov/selinux/index.html.

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

[43] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. Symposium on Operating Systems Design and Implementation, pages 213–227. Usenix Association, Berkeley, CA, 28–31 October 1996.

[44] J. A. Stankovic and K. Ramamritham. The Spring kernel: a new paradigm for real-time operating systems. *Operating Systems Review*, **23**(3):54–71, July 1989.

[45] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **27**(5):203–216. ACM, 1993.

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

[47] Y. Yokote. The Apertos reflective operating system: the concept and its implementation. Object-Oriented Programming: Systems, Languages, and Applications, pages 414–434, 1992.