Usenix Annual Technical Conference, San Antonio, TX, June 9-14, 2003.

System Support for Online Reconfiguration

Craig A. N. Soules † Jonathan Appavoo ‡ Kevin Hui ‡ Robert W. Wisniewski §
Dilma Da Silva § Gregory R. Ganger † Orran Krieger § Michael Stumm ‡
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 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 different implementations that have the same interfaces. This paper describes support for online reconfiguration in the K42 operating system and our initial experiences using it. It describes four base capabilities that are combined to implement generic support for interposition and hot-swapping. As examples of its utility, the paper describes some performance enhancements that have been achieved with K42's online reconfiguration mechanisms including adaptive algorithms, common case optimizations, and workload specific specializations.

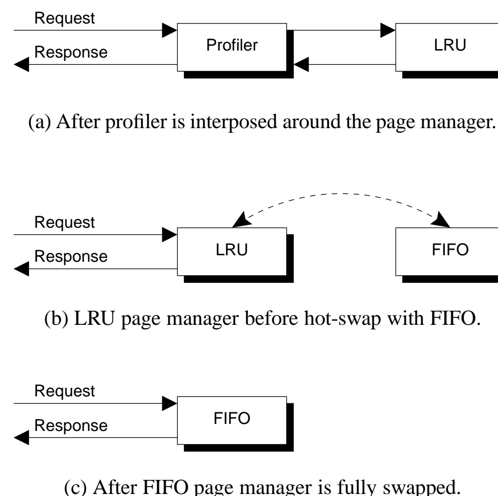


Figure 1: **Online reconfiguration.** This figure shows two online reconfiguration mechanisms: interposition and hot-swapping. (a) shows an LRU page manager and an interposed profiler that can watch the component's calls/returns to see how it is performing. If it determines that performance is poor, 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. Often 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 the above requirements is a need to modify system software after it has been deployed. In some cases, 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, activities that require system modification could be addressed 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 when desired without common-case overhead.

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.

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

To implement interposition and hot-swapping are implemented using 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. Hot-swapping consists of exercising the four capabilities in sequence. Interposition relies mainly on the fourth capability to redirect external references to the interposed code.

This paper discusses these four capabilities in detail, describes our initial prototype implementation of interposition and hot-swapping in the K42 operating system [32], and discusses how online reconfiguration could be added to more traditional operating systems. To explore some of the flexibility and value of online reconfiguration, we implemented a number of well-known dynamic performance enhancements. These include common-case specialization, workload-based specialization, and scalability specialization.

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 case-by-case mechanisms are used for each example.

This section motivates 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

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 and does not place additional requirements on the clients of the component.

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.

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. Although in some cases implementing such an adaptive algorithm may be simple, this approach allows adaptive algorithms to be updated and expanded while the system is running. Each independent algorithm can be developed as a separate component, hot-swapped in when appropriate. Also, interposed code can perform the monitoring, allowing easy upgrades to the monitoring methodology and paying performance penalties only during sampling. Section 6.3 evaluates using online reconfiguration to provide adaptive page replacement.

Specializing the common case: For many individual algorithms, the common code path is simple and can be implemented efficiently. However, supporting all of the complex, uncommon cases often makes the implementa-

tion difficult. 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 [40]. Section 6.3 evaluates the use of online reconfiguration to specialize exclusive access to a file, while still supporting full sharing semantics when necessary.

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 [18, 20, 22, 53]. 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 [28, 48]), 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 [36].

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 Online Reconfiguration Support

This section discusses four main requirements of online reconfiguration. First, components must have well-defined interfaces that encapsulate their 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, because 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. Although it may be possible to detect some rule violations using code analysis [17], any solution still relies on developer diligence and adherence to the programming discipline.

3.2 Quiescent States

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.

Operating systems are often characterized as event driven. The majority of activity in the OS can be represented and serviced as individual requests with any given request having an identifiable start and end. This nature can be leveraged to implement a number of interesting synchronization algorithms. By associating idempotent changes of system data structures with an epoch of requests, one can identify states in which a data structure is no longer being referenced. For example, to swing the head pointer of a linked list from one chain of nodes to another, one can divide the accesses to the list into two epochs. The first epoch includes all threads in the system that were active before the swing, and the second epoch includes any new threads begun after the swing. Because new threads will only be able to access the new chain of nodes, nodes of the old chain are guaranteed to no longer be in use once the threads in the first epoch have ended. At this point, the old chain is quiescent and can be modified at will (including being deleted).

We have utilized this style of synchronization, termed *Read-Copy-Update(RCU)*, to implement a semi-automatic garbage collector [21] and hot-swapping in K42. Others have used it in PTX [37] and in Linux to implement a number of optimization such as: lock-free module loading and unloading [36, 44, 45].

The key to leveraging *RCU* techniques is being able to divide the work of the system into short-lived "requests" that have a cheaply identifiable start and end. In non-preemptive systems such as PTX and Linux[1], a number of key points (e.g., system calls and context switches) can be used to identify the start and end of requests. K42 is preemptable and has been designed for the general use of RCU techniques by ensuring that all system requests are handled on short-lived system threads. As such, thread creation and termination can be used to identify the start and end of requests. Section 4 describes how K42's thread generation mechanism is used to determine quiescent states.

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.

[1]Schemes for extending the current RCU support in Linux to preemptive versions have been proposed[36].

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 will often 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, a 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 redirected to 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.

Reference counting and indirection are two common ways to handle this. Reference counting, as is used in garbage collection [6, 12, 26], tracks all references to a component (even within a single client). If a component changes, all tracked references are found and updated. The drawback of reference counting is that its overhead grows linearly with the number of component references. On the other hand, indirection requires that all references point to a single indirection pointer. To update all of a client's references to an object, the system only needs to update the single indirection pointer for that client. Similarly, if a globally accessible object is being swapped, the indirection pointer of each client using the object must be updated. This is space and performance efficient, with small constant overhead per component used in each client. The drawback is that the programmer must be aware of the indirection and account for it, while reference counting is handled transparently.

3.5 Orthogonal safety issues

The focus of this work is on the mechanics of online reconfiguration. There are orthogonal issues of safety and

security related to deciding which reconfigurations to permit [7, 41] and containing suspect extensions [50, 52]. Other researchers have addressed these issues with several different proposals. The implementation described in Section 4 makes no guarantees about the safety of a reconfiguration. However, most prior techniques for ensuring safety (see Section 7) could be applied to this implementation.

4 Online Reconfiguration in K42

This section describes the integration of online reconfiguration into the K42 operating system. It overviews K42, describes the features used, 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 supports the Linux API and ABI [3] allowing it to use unmodified Linux applications and libraries. 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++) objects [5]. 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 object-oriented approach naturally maps each system component onto a C++ language object. K42 requires that the external interface of a component is defined as a virtual base class for all implementations of the component. This programmer convention enforces the required component boundaries without significant burden on developers.

Quiescent States: K42 employs a technique similar to that discussed in Section 3 to establish quiescent states for an object. In K42, all system requests are serviced by a new system thread. Hence, requests can be divided into epochs by partitioning the threads. Specifically, it is possible to determine when all threads that were in existence on a processor at a specific instance in time have terminated. K42 maintains two thread generations to which threads are assigned.[2] Each generation records the number of threads that are active and assigned to it. At any given time, one of the generations is identified as the current generation and all new threads are assigned to it. To determine when all the current threads have terminated, the following algorithm is used:

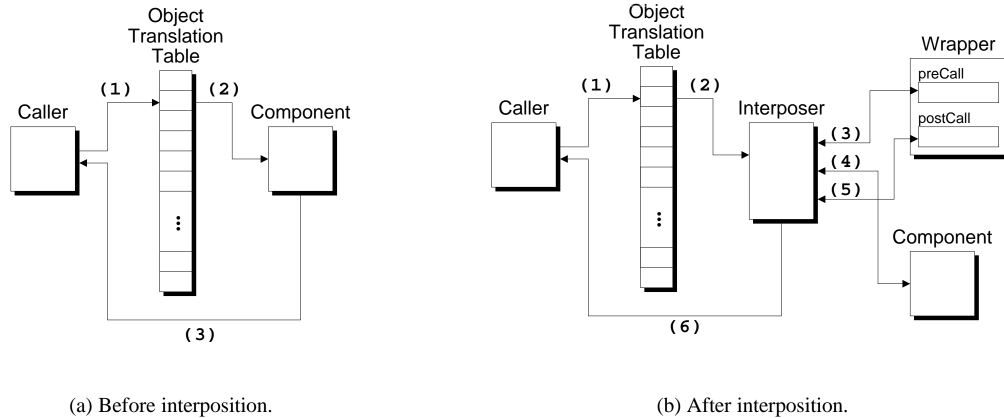
```
i := 0
while (i < 2)
   if (non-current generation's count = 0)
      make it the current generation
   else
      wait until it is zero and
      make it the current generation
   i := i+1
```

In K42, the process of switching the current generation is called a generation swap. The above algorithm illustrates that two generation swaps are required to establish that the current set of threads have terminated. This mechanism is timely and accurate even in the face of preemption, because K42's design does not use long-lived system threads nor does it rely on blocking system-level threads. Note that in the actual implementation, the wait is implemented via a call-back mechanism, avoiding a busy wait.

State transfer: K42 leaves the implementation of individual state transfer methods up to the developer. However, to assist state transfer negotiation, 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, K42 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 the per-client *object translation table* to provide a layer of indirection for accessing system components. When an object instance is

[2] The design supports an arbitrary number of generations but only two are used currently.



(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.

created, an entry for it is created in the object translation table of clients that are accessing it, 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 the entry in the appropriate tables. Although this incurs an extra pointer dereference per component call, the object translation table has other benefits (e.g., improved SMP scalability [21]) that outweigh this overhead.

4.3 Online reconfiguration

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

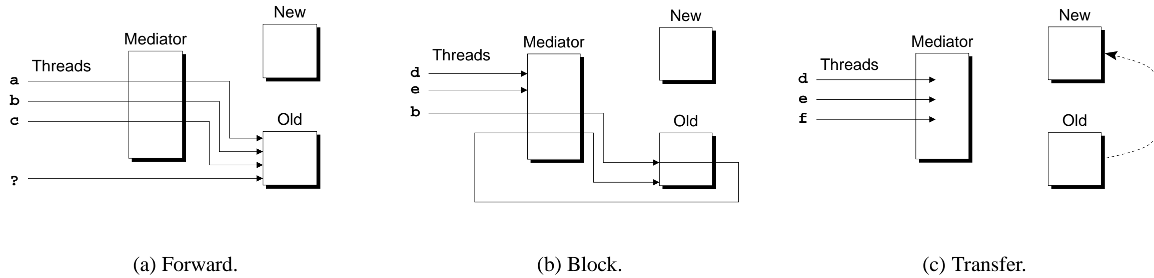
Object interposition interposes additional functionality around all function calls to an existing object instance. We partition interposed functionality into two pieces: a *wrapper object* and a *generic interposer*. The wrapper object contains the specific code that should be executed before and/or after a call to an object. The generic interposer wraps the interface of any object, transparently calling the functions of a given wrapper object before forwarding the call to the original object.

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.

To redirect calls from the original object to the generic interposer, the interposer replaces itself for the object in the object translation table. 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, as shown in Figure 2. It also determines which of the original methods was called, allowing method specific wrappers.

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, because it must be saved by the callee for the original caller.

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 inter-



(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.

poser'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 specialized interposer's internal interfaces. Unfortunately, this is difficult, because they would be in compiler-specified locations that are difficult to determine at run-time. By separating the wrapper, we can avoid specializing the interposition method for each wrapper, because the PRECALL and POSTCALL can be located using the wrapper's virtual function table.

To detach an interposed wrapper, the corresponding interposer object 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 to the original component. This phase continues until all calls started before call tracking began have completed. To detect this, K42 relies on the thread generation mechanism. When the forward phase begins, a request to establish a quiescent state with respect to the current threads is made to the generation mechanism. After two generation swaps, the generation mechanism calls back, because all current threads are guaranteed to have terminated.

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 `?`. While waiting for a quiescent state, new calls `a`, `b`, and `c` are tracked by the mediator. The next phase begins once the generation mechanism indicates that a quiescent state has been achieved.

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, because 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.

Although it would be simpler to combine the forward and blocked phase (blocking new calls immediately and waiting for then generation mechanism to indicate a quiescent state), this would require blocking until all threads in the system have completed. Due to K42's event driven model, thread lifetimes are guaranteed to be short; however, blocking an overloaded component while waiting for every thread in the system to expire may reduce component availability and overall system performance. By separating the phases, blocking is only dependent upon the lifetime of threads active in that 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 (because 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 is detached and the old component destroyed.

4.4 Summary

K42's online reconfiguration mechanisms handle 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. Although not described, considerable effort has gone into ensuring that the interposition and hot-swapping mechanisms are scalable and efficient on a multiprocessor; see [4] for details.

5 Support in Other Systems

This section discusses generally how online reconfiguration can be supported in systems other than K42 and specifically the changes and additions that would be required to add support into Linux.

5.1 Supporting Online Reconfiguration

Many of the support mechanisms used to provide online reconfiguration are useful for other reasons, and were implemented in K42 before online reconfiguration was considered. For similar reasons, many of these support mechanisms already exist in many systems.

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 become eligible for online reconfiguration.

PTX and Linux already have explicit support for RCU techniques, as described in Section 3. Hence, establishing the quiescent state required for hot swapping is straightforward. RCU support can be utilized to implement a number of lock-free synchronization algorithms beyond just hot-swapping. In PTX and Linux RCU, support was added to facilitate several independent lock-free optimizations. But, perhaps more importantly, the use of RCU in PTX and Linux illustrates the fact that the RCU approach used in K42 for hot-swapping generalizes and is equally applicable in traditional systems. This is not surprising, because the event-driven nature of operating systems is widely accepted and is the fundamental property on which RCU relies.

Because indirection leads to added flexibility, most systems have several points of indirection built-in. For example, the indirection used in the VFS layer of many operating systems abstracts the underlying file system implementation from the rest of the system, isolating the other components of the system from the internals of the various file systems. Other examples of indirection include device drivers and virtual memory systems.

In K42, rather than having many different styles or points of indirection, a standard level of indirection was introduced in front of all objects. Directing every object access through an indirection allows K42 to support complex multiprocessor optimizations while preserving a simple object-oriented model [21]. However, it also has allowed us to entertain hot-swapping any instance of an object. Although existing systems might not have such a uniform model of indirection, hot-swapping could first be applied to the components which do live behind a level of indirection, such as VFS modules. If the flex-

ibility of an indirection proves to be useful, one would expect more and more components to utilize it. In time, perhaps standard support for accessing all components behind a level of indirection, similar to K42, would be employed.

Although the state transfer protocol has no clear 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 as straightforward as adding it to K42 was.

5.2 Online Reconfiguration in Linux

This section examines each of the four requirements for online reconfiguration and discusses how to use existing mechanisms and add additional mechanisms to Linux to support online reconfiguration.

Component boundaries: Although Linux is not strictly modular in design, interface abstractions have been added to support loadable modules in many places. These well-defined abstractions could be used to provide the component boundaries required by online reconfiguration.

Quiescent States: Over the last few years, a number of patches have been developed for Linux [36] to add support for RCU mechanisms. This support has been leveraged for a number of optimizations [44] including: adding lock-free lookups of the *dentry* cache, supporting hot-plugging of CPUs, safe module loading and unloading, scalable file descriptor management, and lock-free lookups in the IPV4 route cache. The Linux 2.5 kernel has recently integrated this support into the main line version [45]. This same RCU infrastructure can be utilized to identify the necessary quiescent state for hot-swapping, as is done in K42.

State transfer: Although the boundaries for module state are not as well-defined in Linux as they are in K42, state transfer should be very similar. Also, the same transfer negotiation protocol used by K42 could be applied to Linux.

External references: In Linux, the same module abstraction used to provide component boundaries could also be used to handle external references. Because modules would lie behind a virtual interface, updating the pointers behind this interface is all that is required to replace a given module. For example, replacing a file system module (or individual pieces of file system functionality) could be done by replacing the appropriate function pointers for that file system in the VFS layer.

6 Evaluation

In this section, we evaluate the flexibility and performance of K42's online reconfiguration. First, we quantify the overheads and latencies of interposition and hot-swapping. Second, we illustrate the flexibility of online reconfiguration by using it to implement a number of well-known dynamic performance enhancements.

6.1 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 [33]. 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,[3] 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.

6.2 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 and C++ virtual function table. The overhead of the object translation table is a single memory load, and this data is likely to be cached for frequently

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

Operation	μseconds
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 the cost to component callers zero (because they do not have any additional wait time); however, the process performing the detach must pay the given overhead for destroying the objects. The average cost of each operation is listed in microseconds along with its standard deviation.

accessed objects. The overhead of dispatching a virtual function call is approximately 10 cycles. 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.

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. Because the empty wrapper performs no operations (simply returning from the PRECALL and POSTCALL), all of the call overhead is due to interposition.

Table 1 lists 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, because 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 (because threads are likely to block, increasing the duration of the mediation phases), it is important to understand this sort of "worst-case" swapping scenario.

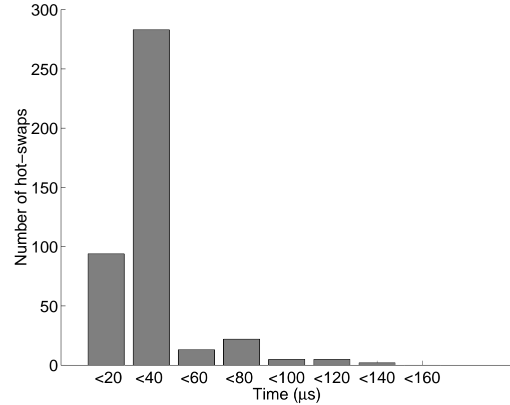


Figure 4: **Null swap.** This figure presents a histogram showing the cost of performing a null-swap of the FCM module at contended points of system execution. For each bin, there is a count of the number of swaps with completion times that fell within that bin. On average, a swap took 27.6 μs to complete, and no swap took longer than 132.6 μs.

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 27.6 μs with a 19.8 μs standard deviation, a 1.06 μs minimum and a 132.6 μs maximum. Hot-swapping while the system is not under contention is more efficient, because 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 10.8 μs. Because most of the time spent doing a hot-swap is spent waiting for the generation to advance (during which no threads are blocked) the affect of these hot-swaps on the throughput of SDET is negligible.

6.3 Reconfiguration for performance

This section evaluates online reconfiguration's flexibility by using it to implement four well-known adaptive performance enhancements. Because these algorithms are well-known, the focus of this section is not on how the adaptive decisions were made, but rather the fact that online reconfiguration can be used to quickly and efficiently implement each of them.

Single v. Replicated FCMs: 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,

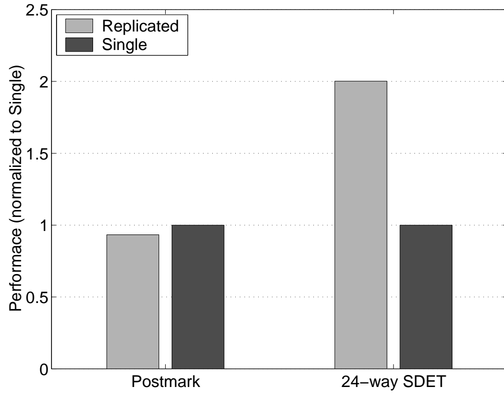


Figure 5: **Single v. Replicated FCMs.** This figure shows two different FCM implementations run under different workloads. In postmark, the single-access FCM performs better because it has less memory overhead during the creation and deletion of files. Conversely, the replicated FCM performs better under 24-way SDET because it scales well to multiple processors.

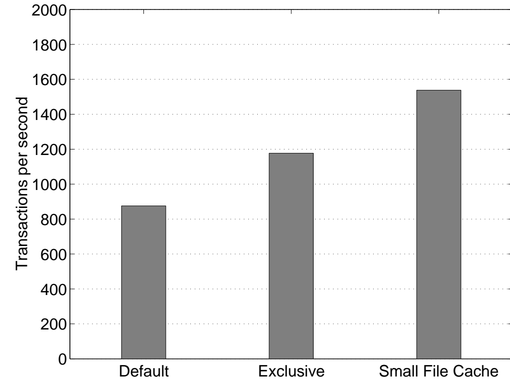


Figure 6: **Common-case optimization.** Using online reconfiguration, K42 can cache exclusive file handles within the application and swap to a shared implementation when necessary for correctness. A further enhancement is to cache small, exclusive files within the application's address space. This figure 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.

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.

Exclusive v. Shared File Sessions: This experiment uses online reconfiguration to swap between an efficient non-shared component and a default shared component for correctness. We used 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 shared access version only when necessary in Postmark. Because most of the accesses in Postmark are exclusive, a 34% performance improvement is achieved.

Small v. Large Files: This experiment 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 improved 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 above, 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. Anecdotally, we found that reimplementing this using online reconfiguration simplified and clarified the code, and it was less time-consuming to implement and debug.

Adaptive Page Replacement: This experiment uses online reconfiguration to implement an adaptive page replacement algorithm. An interposed wrapper object watches an FCM for sequential page mappings. If a sequence of more than 20 pages are requested, the access is deemed sequential, and the system hot-swaps to a sequentially optimized FCM that approximates MRU page replacement. If this sequential behavior ends (more than 10 non-sequential pages are requested), the FCM is swapped back to the default FCM.

Figure 7 shows the performance of 1-way SDET in the

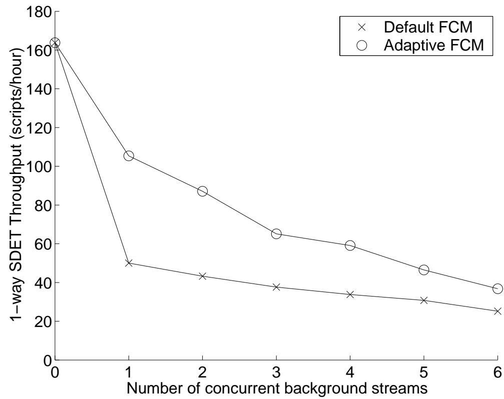


Figure 7: **Sequential page faults.** One adaptive page replacement algorithm performs MRU page replacement for sequential streams. This reduces the amount of memory wasted by streams whose pages will never be accessed again. Using online reconfiguration, K42 can detect sequential streams and swap to 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.

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, because 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.

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 [24]. When a shared library is updated, all programs dependent on the library are automatically updated when they are next run. 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.

Although several online reconfiguration methods exist in the middleware community [34], many do not trasfer well to the realm of OSes due to their increased constraints on timing and resources. Identifying and implementing an online reconfiguration mechanism that works well within an OS is one of the major contributions of this work.

Hjálmtÿsson and Gray describe a mechanism for dynamic C++ objects [31]. 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 [8], DCE [42], RMI [46], and COM [13] 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.

Pu, et al. describe a “replugging mechanism” for incremental and optimistic specialization [43], 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 [38].

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, 25, 35]. Extensible operating systems have shown performance benefits for a number of interesting applications [7, 19, 49]. Technologies such as compiler-directed I/O prefetching [10] and storage latency estimation descriptors [39] improve application performance using detailed knowledge about the state of system structures. Incremental and optimistic specialization [43] 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 [27, 47], extensibility [11], reflection [54], persistence [15], and decentralization [1, 51]. In addition, K42’s method of detecting a quiescent state is not unique. Sequent’s NUMAQ used a similar mechanism for detecting quiescent state [37], and recently, SuSE Linux 7.3 has integrated a mechanism for detecting quiescence in kernel modules [36].

7.1 Open Issues

Although our prototype provides a solid base for numerous OS enhancements, there are a number of open issues that could expand the utility of K42's online reconfiguration. This section describes a number of these issues, and how they have been addressed in other systems.

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. A common solution to this is an object "factory" that is responsible for creating and managing objects in the system. When an upgrade is requested, the factory is responsible for locating and upgrading each instance. Another similar solution is to use a garbage collector to track and update objects [23].

Coordinated swapping: While hot-swapping individual components can provide several benefits, there are times when two or more components must be swapped together. For example, an architecture reconfiguration that updates the interfaces between two objects must swap the objects concurrently. Several different groups have looked into how to perform this while ensuring correctness [9, 14, 30].

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, or show type-safety [23], component validity has been proven to be generally undecidable [29]. Because of this, systems must make a tradeoff between flexibility and provable component correctness.

The most common method for improving component correctness is through type safety. Unfortunately, this generally reduces the flexibility or performance of potential reconfigurations. Dynamic ML [23] provides type safety guarantees for components with unchanging external interfaces, but, its reliance upon a two-space copying garbage collector makes it unreasonable for use in an operating system environment. Hicks [30] provides type safety guarantees for components with external data and changing interfaces, but relies upon programmer defined "safe" swap-points and requires that every instance of a component be swapped (thus two versions of a component cannot exist within the system at once).

Interface management: Currently, K42's online reconfiguration requires that swapped components support the same interface. While it is straightforward to expand these interfaces (because the old interface is a subset of the new interface), it is currently not possible to reduce interfaces in K42; in particular, it is not possible to know if an active code path in the system relies upon a particular part of the interface. Several systems have looked into allowing interface changes using type-safety and programmer defined safe-points for updates to provide the necessary information about component usage [16, 23, 30].

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. Because this becomes increasingly complex as the number of implementations increases, it would be ideal if the infrastructure could 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. For example, Hicks examines the possibility of automatically generating state transformation functions for simple cases [30].

Avoiding quiescence: It may be possible to instantiate the new object and have it start processing calls while the old object completes calls that are still in-flight. In some cases this would require a way to maintain coherence between the states of the two objects; however, in other cases it may be possible to do a lazy update of the new object's state after in-flight calls have completed.

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 used 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.

Acknowledgements

A large community has contributed to the K42 project over the years, and we would like to thank everyone that helped us get the system to the state where the research presented in this paper was possible. Craig Soules was supported by a USENIX Fellowship. Jonathan Appavoo was supported by an IBM Fellowship.

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] J. Appavoo, M. Auslander, D. Edelsohn, D. D. Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Providing a Linux API on the scalable K42 Kernel. Page to appear.

[4] J. Appavoo, K. Hui, M. Stumm, R. W. Wisniewski, D. D. Silva, O. Krieger, and C. A. N. Soules . An infrastructure for multiprocessor run-time adaptation. ACM SIGSOFT Workshop on Self-Healing Systems. ACM, 2002.

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

[6] 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.

[7] B. N. Bershad, S. Savage, P. Pardyn, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5), 3–6 December 1995.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] R. P. Cook and I. Lee. DYMOS: A dynamic modification system. Pages 201–202.

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

[16] D. Duggan. *Type-based hot swapping of running modules*. Technical report SIT–CS–2001–7. October 2001.

[17] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. ACM Symposium on Operating System Principles. ACM, 2001.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] S. Gilmore, D. Kirli, and C. Walton. *Dynamic ML without dynamic types*. Technical report ECS–LFCS–97–378. June 1998.

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

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

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

[27] 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.

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

[29] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, **22**(2):120–131. IEEE, February 1996.

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

[31] 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.

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

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

[34] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, **45**(6):33–38. ACM Press.

[35] 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.

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

[37] 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.

[38] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems (TOCS)*, **19**(2):217–251. ACM Press.

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

[40] 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.

[41] 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.

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

[43] 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.

[44] Read-Copy Update Mutual Exclusion for Linux, http://lse.sourceforge.net/locking/rupdate.html.

[45] Posting by Linus Tovalds to linux-kernel mailing list: Summary of changes form v2.5.42 to v2.5.43, http://marc.theaimsgroup.com/?l=linux-kernel&m=103474006226829&w=2.

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

[47] 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.

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

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

[50] 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.

[51] 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.

[52] 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.

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

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