Appeared in Proc. of the first ACM Workshop on Self-Healing Systems, Charleston, SC, USA, November 2002, pp. 3-8.

# An Infrastructure for Multiprocessor Run-Time Adaptation

Jonathan Appavoo, Kevin Hui, and Michael Stumm
University of Toronto
jonathan,khui,
stumm@eecg.toronto.edu

Robert W. Wisniewski, Dilma Da Silva, and Orran Krieger
IBM T. J. Watson Research Center
bob,dilma,okrieg@watson.ibm.com

Craig A. N. Soules
Carnegie Mellon University
soules@cmu.edu

## 1. INTRODUCTION

Runtime adaptation and dynamic reconfiguration allow a system to dynamically swap in the most appropriate implementation of its components based on current or expected use, and to selectively upgrade components with bug, security, or performance fixes without down-time. *Hot-swapping* is the act of replacing an active system component's implementation with a new or different implementation while maintaining availability of the component's functionality. This paper describes a mechanism to hot-swap software components within the K42[1] operating system. We have used this capability to improve performance under a variety of conditions.

K42 is a research operating system for cache-coherent shared memory multiprocessors designed to achieve good scalability on a wide range of workloads. In K42, each virtual and physical resource, e.g., open file, memory region, page table, is managed by separate, fine-granularity, object instances. Each object instance may be customized (different sub-class). This model provides the standard software engineering benefits, but more importantly: 1) allows performance customization on an object-by-object basis and 2) allows, on a multiprocessor, independent accesses to be directed to independent instances and proceed in parallel thus eliminating shared memory accesses and synchronization, which are fundamental barriers to multiprocessor performance.

An operating system is a particularly demanding environment as it needs to satisfy multiple concurrent clients whose resource demands may be at odds. For example, multiple clients may simultaneously access a file with different usage patterns. Supporting multiprocessors presents additional challenges for operating systems, and often implementations that are required for scalable performance have worse uniprocessor behavior.

To provide both uniprocessor and multiprocessor components within a single object model, K42's objects are implemented using a novel design called *Clustered Objects*. A Clustered Object can be internally decomposed into a group of cooperating constituent subparts that implement a uniform interface, but use distributed structures and algorithms to avoid shared memory and synchronization on its frequent and critical operations. Clustered Objects provide an infrastructure to implement both shared and distributed (across shared-memory multiprocessor) objects, and use, transparently to the client, the implementation appropriate for the access pattern of the object.

This research work applies to shared memory multiprocessors. Our use of the word distributed throughout this paper refers to the division of data across a multiprocessor complex. Distribution does not imply message passing rather, *distributed* data is accessed via hardware supported shared memory. We *distribute* objects in order to: (*i*) optimize cache line access, (*ii*) increase concurrency and (*iii*) exploit local memory on NUMA multiprocessors.

To hot swap code, the system must be able to identify and encapsulate the code and data for a swappable component. An object-oriented model provides this. Clustered objects add efficient multiprocessor capability to this model. By providing both a shared implementation and a scalable implementation for objects, and hot swapping between them, K42 can obtain the best performance characteristics of both implementations depending on usage.

To perform a successful hot swap, it is necessary to get the target object into a *quiescent state*, a point when it is safe to swap implementations. Then all external references (both data and code pointers) to the swapped object need to be modified. In a scalable system, such as K42, components are implemented with a high degree of concurrency. This makes hot swapping more difficult than in a traditional system because it is necessary to coordinate all processors running parts of the to-be-swapped object.

This paper focuses on mechanisms, describing Clustered Objects and an algorithm performing hot-swapping. The important issue of policy, namely the control of when and what objects to hot swap are the subject of future papers.

The remainder of this paper is organized as follows. Section 2 describes K42's Clustered Object implementation. Section 3 describes K42's hot-swapping support. Section 4 provides an example of an existing performance improvement implemented by hot-swapping Clustered Objects and describes the current status of K42's implementation.

[1] http://www.research.ibm.com/K42

## 2. K42 COMPONENT MODEL

One of K42's primary goals is to realize near perfect scalability on a diverse set of workloads. This goal has fundamentally influenced its design. A key to achieving high performance on a multiprocessor is to use per-processor data structures whenever possible, so as to minimize inter-processor coordination and shared memory access. An example is a simple performance counter where the predominant action on the counter is to increment it. On a multiprocessor, there is a large benefit to using a distributed counter rather than a single shared one. In the distributed case, each processor increments a per-processor sub-counter that does not require synchronization or shared memory access. When the value of the counter is required, the per-processor sub-counters are summed.

When implementing a distributed counter, there are a number of aspects that are desirable. The distributed nature of the counter should be hidden behind its interface, preserving clean component boundaries and isolating client code from the distribution. An instance of the counter should have a single, unique, processor-independent identifier, that transparently directs the caller to the correct processor specific sub-counter. Care must be used to ensure that the process for directing an access to a sub-counter in the common case does not require shared structures or incur significant costs. Using shared structures to access a per-processor subcounter would defeat the purpose of trying to eliminate sharing. Further, if the costs to direct an access to the correct sub-counter are too expensive, then the use of this approach becomes questionable when the counter is only accessed on a single processor. To ensure scalability on systems with a large number of processors, a lazy approach to allocating the sub-counters is necessary. This ensures that the costs of allocating and initializing the sub-counter only occur on processors that access the counter.

One of the key features of K42's scalability is a unique distributed component model that facilitates the development of distributed components and addresses the issues of the previous paragraph. These components, called Clustered Objects, support distributed designs while preserving the benefits of a component based approach. Collections of C++ classes are used to define a Clustered Object and runtime mechanisms are used to support the dynamic aspects of the model. Clustered Objects are conceptually similar to design patterns such as facade, however, they have been carefully constructed to avoid any shared front end, and are primarily used for achieving data distribution. Some distributed systems [2, 5] have explored similar object models.

Each Clustered Object is identified by a unique interface that every implementation conforms to. We use a C++ pure virtual base class to express a component interface. An implementation of a component consists of two portions: a Representative definition and a Root definition, expressed as separate C++ classes. The Representative definition of a Clustered Object defines the per-processor portion of the Clustered Object. In the case of the performance counter, it would be the definition of the sub-counters. An instance of a Clustered Object Representative class is called a Rep of the Clustered Object instance. The Representative class implements the interface of the Clustered Object, inheriting from the Clustered Object's interface class. The Root class defines the global portions of an instance of the Clustered Object. Every instance of a Clustered Object has exactly one instance of its Root class that serves as the internal central anchor or "root" of the instance. Each Rep has a pointer to the Root of the Clustered Object instance it belongs to. As such, the methods of a Rep can access the shared data and methods of the Clustered Object via its root pointer.

At runtime, an instance of a given Clustered Object is created by instantiating an instance of the desired Root class.[2] Instantiating the Root establishes a unique *Clustered Object Identifier* (COID), that is used by clients to access the newly created instance. To the client code, a COID appears to be a pointer to an instance of the Rep Class. To provide better code isolation, this fact is hidden from the client code with the macro: `#define DREF(coid) (*(coid))`. For example, if `c` is a variable holding the COID of an instance of a clustered performance counter, that has a method `inc`, a call would look like: `DREF(c)->inc()`.

A set of tables and protocols are used to translate calls on a COID in order to achieve the unique runtime features of Clustered Objects. There is a single shared table of Root pointers called the Global Translation Table and a set of Rep pointer tables called Local Translation Tables, one per processor. The virtual memory map for each processor is set up so that a Local Translation Table appears at address `vbase` on each processor but is backed by different physical pages[3]. This allows the entries of the Local Translation Tables, which are at the same virtual address on each processor, to have on different values on each processor. Hence, the entries of the Local Translation Tables are per-processor despite only occupying a single range of fixed addresses. When a Clustered Object is allocated, its root is instantiated and installed into a free entry in the Global Translation Table. The Translation Table entries are managed on a per-processor basis by splitting the global table into per-processor regions of which each processor maintains a free list and only allocates from its range, avoiding synchronization or sharing. The address of the corresponding location in the Local Translation Tables address range is the COID for the new Clustered Object. The sizes of the global and local entries and tables are kept the same, allowing simple pointer arithmetic to convert either a local to global or global to local table pointer. Figure 1 illustrates a Clustered Object instance and the translation tables.

To achieve the lazy creation of the Reps of a Clustered Object, Reps are not created or installed into the Local Translation Tables when then Clustered Object is created. Instead, the entries of the Local Translation Table that have not been accessed on a processor are initialized with a pointer to a special object called the Default Object. The Default Object leverages the fact that every call to a Clustered Object goes through a virtual function table[4]. The Default Object overloads the method pointers in its virtual function table to point at a single trampoline method. The trampoline code saves the current register state on the stack, looks up the Root installed in the Global Translation Table entry cor-

[2]The client code is not actually aware of this fact. Rather, a static *Create* method of the Rep class is used to allocate the root. Because we do not have direct language support, this is a programmer enforced protocol.

[3]In K42, a page table is maintained per-processor per-address space, and thus each processor can have its own view of the address space.

[4]As noted above, a virtual base class is used to define the interface for a Clustered Object

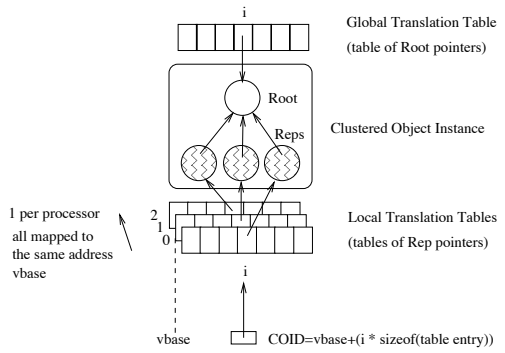



**Figure 1: A Clustered Object Instance and Translation Tables.**

responding to the COID that was accessed, and invokes a well-known method that all Roots must implement called `handleMiss`. This method is responsible for installing a Rep on the processor into the Local Translation Table entry corresponding to the COID that was accessed. This is done either by instantiating a new Rep or identifying a preexisting Rep and storing its address into the address pointed to by the COID. On return from the `handleMiss` method, the trampoline code restarts the call on the correct method of the newly installed Rep. The above process is called a Miss and its resolution Miss-Handling. Note that after the first Miss on a Clustered Object instance, on a given processor, all subsequent calls on that processor will proceed as standard C++ method invocations via two pointer dereferences. Thus, in the common case, methods of the installed Rep will be called directly with no involvement of the Default Object.

The map of processors to Reps is controlled by the Root Object. A shared implementation can be achieved with a Root that maintains one Rep and uses it for every processor that accesses the Clustered Object instance. Distributed implementations can be realized with a Root that allocates a new Rep for some number (or Cluster) of processors and complete distribution is achieved by a Root that allocates a new Rep for every accessing processor. There are standard K42 Root classes that handle these scenarios. In the case of the distributed versions, the Clustered Object implementor defines a new Root class by inheriting from one of the standard distributed Root classes, adding any shared data and methods to it as necessary.

# 3. HOT-SWAPPING

There are a number of challenges in the design of hot-swapping infrastructure capable of dynamically switching a "live" or "hot" software component: 1) avoid adding overhead to normal method invocations, 2) avoid complicating the design of the objects that have switch capabilities, 3) ensure the switching code is scalable, 4) correctly handle in-flight requests to the object being switched, 5) avoid deadlock during the switch, and 6) guarantee integrity when transferring state from the old to the new object. The distributed nature of Clustered Objects further exacerbates these challenges as it can mean having to swap the multiple constituents of a component across multiple processors in a coordinated way.

## 3.1 Overview

Our swapping mechanism allows any Clustered Object instance to be hot-swapped with any other Clustered Object instance that implements the same interface. Moreover, swapping is transparent to the clients of the component and thus no support or code changes are needed in the clients.

### *3.1.1 Algorithm Overview*

The outline of our algorithm is as follows (and described in more detail further below):

(*i*) instantiate the replacement Clustered Object instance;
(*ii*) establish a quiescent state for the instance to be replaced so that it is temporarily idle;
(*iii*) transfer state from the old instance to the new instance;
(*iv*) swap the new instance for the old, adjusting all references to the instance; and
(*v*) deallocate the old instance.

There are three key issues that need to be addressed in this design. The first, and most challenging issue, is how to establish a quiescent state so that it is safe to transfer state and swap references. The swap can only be done when the instance state is not currently being accessed by any thread in the system. Perhaps the most straightforward way to achieve a quiescent state would be to require all clients of the Clustered Object instance to acquire a reader-writer lock in read mode before any call to the object (as done in the re-plugging mechanism described in [7]). Acquiring this external lock in write mode would thus establish that the object is safe for swapping. However, this approach adds overhead for the common case and can cause locality problems, defeating the scalability advantages of Clustered Objects. Further, the lock could not be part of the component itself and the calling code would require changes. Our solution avoids these problems and is presented in the next section.

The second issue is deciding what state needs to be transferred and how to transfer the state from the old component to the new one, both safely and efficiently. We provide a protocol that Clustered Objects developers can use to negotiate and carry out the state transfer. Although the state could be converted to some canonical, serialized form, one would like to preserve as much context as possible during the switch, and handle the transfer efficiently.

The final issue is how to swap all of the references held by the clients of the component so that the references point to the new instance. In a system built using a fully-typed language such as Java, this could be done using the same infrastructure as used by garbage collection systems. However, this would be prohibitively expensive for a single component switch, and would be overly restrictive in terms of systems language choice. An alternative would be to partition a hot-swappable component into a front-end component and a back-end component, where the front-end component is referenced (and invoked) by the component clients and is used only to forward requests to the back-end component. Then there would be only a single reference (in the front-end component) to the back-end component that would need to be changed when a component is swapped, but this adds extra overhead to the common call path. Given that all accesses to a Clustered Object in K42 already go through a level of indirection, namely the Local Translation Table, the more natural way to swap references in our system is to

overwrite the entry pointers in a coordinated fashion. Several distributed systems have examined ways to dynamically configure the location of components, requiring much of the same support [4].

## 3.2 Details

To implement the swapping algorithm outlined above, a specialized Clustered Object called the Mediator Object is used during the swap. It coordinates the switch between the old and new objects, leveraging the Clustered Object infrastructure to implement the swapping algorithm. To handle then swapping of distributed Clustered Object instances with many parallel threads accessing it, the Mediator is itself a distributed Clustered Object that implements the swapping algorithm in a distributed manner by utilizing a set of worker threads.

The Mediator establishes a worker thread and Rep on each processor that the original Clustered Object instance has been accessed on[5] . The Mediator instance is interposed in front of the target Clustered Object instance and intercepts all calls to the original object for the duration of the swapping operation. The details of how the interposition is achieved will be described later. The worker threads and Mediator Reps transit through a sequence of phases in order to coordinate the swap between the old Clustered Object instance and the new one replacing it. The Mediator Reps function independently, only synchronizing when necessary in order to accomplish the swap. Figure 3 illustrates the phases that a Mediator Rep goes through while swapping a Clustered Object. The remainder of this section describes how the Mediator Reps and worker threads accomplish the swap. We present the actions that occur on a single processor but in the general case these actions proceed in parallel on multiple processors.

Prior to initiating the swap (figure 3a) the old object's Reps are invoked as normal. The first step of hot-swapping is to instantiate the new Clustered Object instance, specifying that it not be assigned a COID, and that the installation of its root into the Global Translation Table be skipped. Second, a new Mediator instance is created and passed both the COID of the old instance and a pointer to the Root of the new instance. The Mediator then proceeds to interpose itself in front of the old instance.

Interposing a Mediator instance in front of the old Clustered Object instance ensures that future calls temporarily go through the Mediator. To accomplish this, the Mediator instance must override both the Global Translation Table entry root pointer and all the active Local Translation Table entries' Rep pointers. To swing the Global Translation Table entry Root pointer, it must ensure that no misses to the old object are in progress. As part of the standard Miss-Handling infrastructure, there is a reader-writer lock associated with each Global Translation Table entry and all misses to the entry acquire this lock in read mode. In order to atomically swing the Global Translation pointer, the associated reader-writer lock is acquired for write access, ensuring that no misses are in progress. When the lock has been successfully acquired, the Root pointer of the entry is changed to point to the Root of the Mediator and all future

[5]Our current implementation, as described in this paper, uses per-processor worker threads. We are currently exploring a new implementation that does not require worker threads.

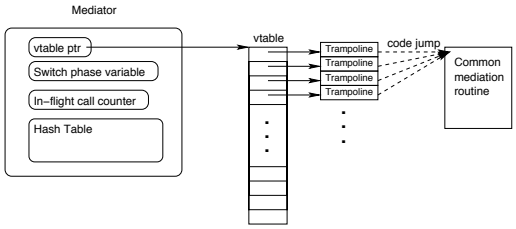



**Figure 2: Mediator Rep implementation**

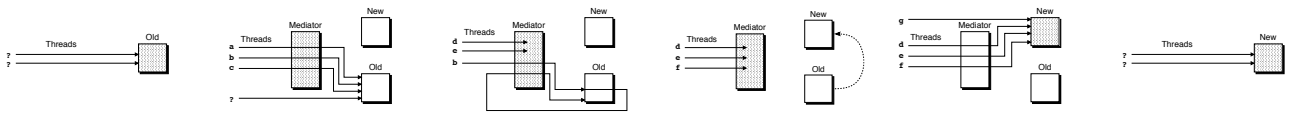
misses will be directed to it. The Mediator remembers the old Object's Root in order to communicate with it. During this process there may be calls that are in flight to the old Clustered Object, and they proceed normally.

Swinging the Root is not sufficient to direct all calls to the Mediator instance. This is because some Rep pointers may already established in the Local Translation Table entry associated with the old instance casuing some calls to proceed directly to the Reps of the old instance. To handle this, the Mediator spawns a worker thread on all the processors that have accessed the old object. These threads have a number of responsibilities, but their first action is to reset the Local Translation entry on each processor back to the Default Object. This ensures that future accesses will be directed to the Mediator Object via the standard Miss-Handling process. Because the Root maintains the set of processors it has suffered a Miss on, the Mediator can query the old object's Root to determine what processors to spawn threads on.

On each Mediator miss, the Mediator Root installs a new Mediator Rep into the Local Translation Table for the processor on which the Miss occurred. The Mediator Reps are specialized C++ objects similar to the Default Object. They are designed to handle hot-swapping of any Clustered Object transparently. To do so, the Mediator Rep intercepts all calls and takes action based on the current phase of the Rep(figure 3).

Figure 3, parts b, c, d and e, illustrate a single Mediator Rep in the different phases of a swap. Once the Mediator Rep has been installed into the Local Translation Table entry, virtual method calls that would normally call one of the functions in the original object end up calling the corresponding method in the mediator. A small amount of assembly glue captures the low-level state of the call, including the parameter passing registers and return address. The actions that the Mediator Rep has to take on calls during the various phases of swapping include: forwarding and keeping a count of active calls (increment prior to forwarding the call and decrement after the forwarded call returns), selectively blocking calls, and releasing previously blocked calls. To be transparent to the clients and the target Rep when the call is being forwarded, the Mediator Rep may not alter the stack layout and hence it must only use Rep-local storage to achieve the appropriate actions. As can be seen in figure 2, the Mediator Rep utilizes three other data members other than its vtable pointer.

The vtable of the Mediator Rep, like that of the Default Object, is constructed to direct all calls regardless of its signature to a single common mediation routine. When a phase requires that new calls be tracked and forwarded, the Mediator Rep uses an in-flight call counter to track the number of live calls. Because the counter needs to be decremented when the call completes, the Mediator must ensure that the forwarded call returns to the mediation routine prior to re-




(a) Prior. (b) Forward. (c) Block. (d) Transfer. (e) Complete. (f) Post.

**Figure 3: Component hot-swapping.** **This figure shows the phases of hot-swapping with respect to a single processor and the Reps: prior, forward, block, transfer, complete, and post. 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.**

turning to the original caller. This means that the Mediator Rep must keep track of where to return to after decrementing its in-flight call counter on a per-thread basis. To maintain transparency it avoids using the stack by maintaining a hash table indexed by thread id to record the return address for a given thread. The Mediator Rep also uses a data member to track the phase it is currently in. The phases are detailed in the following paragraphs.

### *3.2.1* Forward

This initial phase is illustrated in figure 3b. The Mediator stays in this phase until it determines that there are no longer any threads that were started prior to the swap initiation still accessing the object. To detect this, the worker thread utilizes services of K42's Read-Copy-Update[6] mechanism. Specifically, it is possible to determine when all threads in existence on a processor at a specific instance in time have terminated. K42's threads are assigned to one of two generations.[6] 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 the non-current generation's count is zero
        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 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. K42's design does not use long-lived system threads nor does it rely on blocking system-level threads[6]. Note that in the actual implementation, the wait is implemented via a call back mechanism avoiding a busy wait.

By waiting for all threads that were in existence when the swap was initiated to terminate, we are sure that all threads accessing the old object have terminated. However, to ensure system responsiveness while waiting for these threads to terminate, new calls to the object are tracked and forwarded by Mediator Rep using its in-flight call counter and hash table. The thread descriptors are also marked as being in a forwarded call in order to simplify deadlock avoidance as described in the next paragraph. Once the worker thread, using the generation mechanism, determines that all the untracked threads have terminated, the Mediator Rep switches to the *Block* phase. Note that the *Forward* phase, and transition to the *Block* phase, happen independently and in parallel on each processor, and no synchronization across processors is required.

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

### *3.2.2* Block

In this phase, the Mediator establishes a quiescent state, guaranteeing no threads are accessing the old Clustered Object on any processor. To do this, each Mediator Rep establishes a quiescent state on its processor by blocking all new calls while waiting for any remaining tracked calls to complete. However, because a call currently accessing the object might itself call a method of the object again, care must be taken not to cause a deadlock by blocking any tracked threads. This is achieved by checking the thread descriptor to determine if the thread is in a forwarded call. This also ensures that concurrent swaps of multiple Clustered Objects do not deadlock. If a forwarded thread, during the blocked phase in one object, calls another object that is in the blocked phase it will be forwarded rather than blocked, thus avoiding the potential for inter-object deadlocks. To ensure a quiescent state across all processors, the worker threads must synchronize at this point prior to proceeding. A shared data member in the Mediator root is used for this purpose.

### *3.2.3* Transfer

Once the Blocked phase has completed, the Transfer phase begins. In this phase the worker threads are used to export the state of the old object and import it into the new object. To assist state transfer, a *transfer negotiation protocol* is provided. 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 create a prioritized list of the state transfer protocols that the component supports. For example, it may be best to pass internal structures by memory reference, rather than marshaling the entire structure; however, both components must understand the same structure for this to be possible. Before initiating a hot-swap, a list is obtained from both the old and new component instances. The most desirable format,

based on the two lists, is recorded by the Mediator instance. The actual data transfer is carried out in parallel by the worker threads. The worker threads request the state from the old object in the format that was recorded in the Mediator instance and pass it to the new object.

#### *3.2.4* Complete

After the state transfer, the worker threads again synchronize so that one may safely swing the Global Translation Table entry to the Root of the new Clustered Object. All the worker threads then cease call interception by storing a pointer to the new Clustered Object's Reps into the Local Translation Table entries so that future calls go directly to the new Clustered Object. The worker threads then resume all threads that were suspended during the *Block* phase and forward them to the new Object (figure 3e). The worker threads deallocate the original object and the mediator and then terminate.

## 4. STATUS

K42 is under active research and development. It runs on x86-64 and PowerPC simulators, and runs on PowerPC multiprocessor hardware. K42 supports a Linux API; it uses standard GNU packages as its user-level software base. The hot-swapping infrastructure is fully functional with key virtual memory and file system components utilizing it to dynamically switch between implementations based on runtime demands. As an example, the File Cache Manager (*FCM*) virtual memory component has two implementations: shared and distributed. For each open file, an instance of an FCM is used to cache the pages of the file's data in physical frames. By default, to achieve better performance when a file is opened, a simple, shared implementation of the FCM is initially instantiated. If the file is accessed only on one processor, the shared FCM implementation performs well with little memory overheads. When the file is accessed by multiple processors concurrently, the file's associated FCM is hot-swapped to a distributed implementation. This alleviates any contention and achieves better scalability, thus ensuring that only the files that experience contention due to sharing use the more complex and expensive distributed FCM implementation.

The advantages of hot-swapping the FCM implementations became visible when measuring the performance of two different benchmarks: PostMark[3] and Spec SDET[1]. PostMark is a uniprocessor file system benchmark that creates a large number of small files on which a number of operations are performed, including reading and appending. SDET is a multiprocessor Unix development workload that simulates concurrent users running standard Unix commands[7]. If we disable hot-swapping and run PostMark using only shared FCMs, and then run it using only distributed FCMs, there is a 7% drop in performance for the distributed implementation of the FCM. However, running SDET with the distributed FCM implementation, yields an 8% performance improvement on 4 processors and an order of magnitude improvement on 24 processors. When hot-swapping is enabled, the best-case performance is achieved automatically for both PostMark and SDET, with the individual FCM instances choosing the right implementation based on the demands it experiences.

This experiment demonstrates that our hot-swapping mechanism is capable of dealing with complex scenarios under load. The experiment also illustrates that by using hot-swapping we are able to avoid codifying ad hoc and brittle rules i.e., we did not need to hard code the fact that certain files, such as program executables, should use the distributed FCM. Instead, hot-swapping allowed each file to settle on the right implementation based on how it was dynamically accessed.

We are still at the early stages of exploring the uses of runtime adaptation via hot-swapping. At this point we are using simple triggers such as lock contention to prompt switches between implementations. There is much work to be done in the algorithms and in the triggers used to decide when to hot-swap a component and what the new implementation should be. Other open issues include:

- How to express the tradeoffs between implementations in a generic way such that automated methods of choosing the right implementation can be achieved.
- How to coordinate multiple component swaps in order to achieve a global improvement while maintaining stability.
- What kind of language support would be beneficial?

## 5. REFERENCES


[1] SPEC SDM suite. http://www.spec.org/osg/sdm91/, 1996.

[2] P. Homburg, L. van Doorn, M. van Steen, A. S. Tanenbaum, and W. de Jonge. An object model for flexible distributed systems. In *First Annual ASCI Conference*, pages 69–78, Heijen, Netherlands, may 1995.

[3] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance. PostMark: A New File System Benchmark.

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

[5] M. Makpangou, Y. Gourhant, J.-P. L. Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, Los Alamitos, California, 1994.

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

[7] 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, 2001.


[7] We used a slightly modified version that does not include a compiler or ps.