Appeared in Proc. Workshop on Hot Topics in Operating Systems (HotOS),
Elmau/Oberbayan, Germany, May 2001.

Supporting Hot-Swappable Components for System Software

Kevin Hui [†] Jonathan Appavoo [†] Robert Wisniewski [‡]
Marc Auslander [‡] David Edelsohn [‡] Ben Gamsa [§]
Orran Krieger [‡] Bryan Rosenburg [‡] Michael Stumm [§]

Abstract

Supporting hot-swappable components allows components to be replaced even while they may be in active use. This can allow live upgrades to running systems, more adaptable software that can change its behaviour at run-time by swapping components, and a simpler software structure by allowing distinct policy and implementation options to be implemented in separate components (rather than as a single monolithic component) and dynamically swapped as needed.

We describe in general the challenges that must be faced to support hot-swapping, then describe our prototype in the K42 operating system and provide some initial results.

1 Introduction

A hot-swappable component is one that can be replaced with a new or different implementation while the system is running and actively using the component. For example, a component of a TCP/IP protocol stack, when hot-swappable, can be replaced—perhaps to handle new denial-of-service attacks or improve performance—without disturbing existing network connections. Such a capability offers a number of potential advantages:

Increased Uptime: Numerous mission-critical systems require five-nines level uptime, making software upgrades extremely challenging. Support for hot-swappable components would allow software to be upgraded (i.e., for bug fixes, new features, performance improvements, etc.) without having to take the system down. Telephony systems, control systems, operating systems, and database systems are examples of software systems that are used in mission-critical settings and would benefit from hot-swappable component support.

Improved Performance: The best implementation and policy for a given component often depends on how it is used. For example, to obtain good performance in multiprocessor systems (one of our primary research areas), components servicing parallel applications often require fundamentally different data structures compared to those for sequential applications. Concurrently accessed components must support high degrees of concurrency, often at the expense of more complex data structures and longer instruction paths compared to components designed for sequential access. However, when a component is created (for example, when a file is opened), it is generally not known how it will be used. With support for hot-swappable components, a component designed for sequential applications can be used initially, and then swapped for one supporting greater concurrency if contention for the component is detected.

Simplified Software Structure: To support multiple and adaptable policies, a system component generally must implement all alternatives, adjusting its control flow based on the current policy choice or usage. Adding support for debugging and performance monitoring can further complicate the implementation. With hot-swappable components, each policy and option can often be implemented as a separate, independent component, with components swapped as needed. This separation of concerns can greatly simplify the overall structure of the software system.

In order to hot-swap a component, it is necessary to (*i*) instantiate the replacement component, (*ii*) establish a quiescent state in which the component is temporarily idle, (*iii*) transfer state from the old component to the new component, (*iv*) swap the new component for the old, and (*v*) deallocate the old component. In doing so, three fundamental problems need to be addressed:

- The first, and most challenging problem, is to establish a quiescent state when it is safe to

[†] University of Toronto, Dept of Computer Science
[‡] IBM T. J. Watson Research Center
[§] University of Toronto, Dept of Electrical and Computer Engineering

transfer state and swap components. The swap can only be done when the component state is not currently being accessed by any thread in the system. Perhaps the most straightforward way to achieve a quiescent state would be to require all clients of the component to acquire a reader-writer lock in read mode before any call to the component. Acquiring this external lock in write mode would thus establish that the component is safe for swapping. However, this would add overhead in the common case, and cause locality problems in the case of multiprocessors.

- The second problem is transferring state from the old component to the new one, both safely and efficiently. 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 in the face of components with potentially megabytes of state accessed across dozens of processors.

- The final problem is swapping all of the references held by client components so that the references now refer to the new one. In a system built around a single, fully typed language, like Java, this could be done using the same infrastructure as used by garbage collection systems. However, this would be prohibitively expensive for a single component switch, and would be overly restrictive in terms of systems language choice.

We have designed and implemented a mechanism for supporting hot-swappable components that avoids the problems alluded to above. More specifically, our design was driven by the following goals:

- zero performance overhead for components that will not be swapped
- zero impact on performance when a component is not being swapped
- complete transparency to client components
- minimal code impact on components that wish to be swappable
- zero impact on other components and the system as a whole during the swapping operation
- good performance and scalability; that is, the swapping operation itself should incur low overhead and scale well on multiprocessor systems.

Our mechanism has been implemented in the context of the K42 operating system [**?**], in which components in the operating system and in applications that run on K42 have been made hot-swappable. This paper describes our design and implementation, presents preliminary performance numbers with respect to swapping overhead, and illustrates some of the performance benefits such a facility can provide.

2 Design Overview

Our approach to hot-swapping components leverages three key features of our system (their importance will become clear shortly); however, each could be adapted to a more conventional system. First, because K42 is structured in an object-oriented manner using C++ [**?**], a system component maps naturally to a language object,[1] allowing every object to be swapped for another that implements the same interface;[2] a similar approach could be used in a non-object-oriented system that uses operations tables, such as vnodes. Second, in support of a new form of scalable data structure (called *Clustered Objects* [**?**]), K42 provides an extra level of indirection for all major objects; this would need to be added explicitly in other systems lacking it. Finally, the hot-swapping facility targets K42's kernel and system servers, in which each service request is handled by a new, typically short-lived thread (long-lived daemon threads are treated specially); any event-driven system in which control frequently reaches what we call a safe point (such as the completion of a system call, or entering a long term sleep) would suffice.

Given the above, the hot-swapping algorithm is as follows: (*i*) establish a quiescent state for the component; (*ii*) transfer the component state between the old and the new object; and (*iii*) update the references to the component.

To establish a quiescent state, in which it is guaranteed that no threads are currently accessing the object to be swapped, we first atomically swap the indirection pointer so that it points to an interposing object that initially just tracks all threads making calls to the object and passes on the call to the original object. We next wait for the termination

[1] We use the terms *component* and *object* interchangeably throughout the rest of this document.

[2] More specifically, as long as two objects inherit from a common base class that defines the exported interface, we can transparently swap between them by leveraging object polymorphism.

of all calls that were started before call tracking began. We use the short-lived nature of the system threads as a simple way of detecting this point: our system provides an efficient means of determining when all threads that were started before a given point in time have completed (or have reached a safe point). Next, the interposing object temporarily blocks all new calls from proceeding to the original object while it waits for the tracked calls to complete (recursive calls are detected and allowed to proceed, however). Once all the tracked calls have completed, we have reached a quiescent state for the object.

To make state transfer efficient and preserve as much of the original state and semantics as possible, the original and new objects next negotiate a *best common format* that they both support. This, for example, may allow a hash table to be passed directly through a pointer, rather than converted to and from some canonical form, such as a list or array, as well as, in a large multiprocessor, allow much of the transfer to occur in parallel across multiple processors, preserving locality when possible.

Finally, the swap is completed by changing the indirection pointer to refer to the new object, releasing all the threads blocked in the interposing object, and deallocating the original object and the temporary interposing object.

3 Implementation Highlights

One of the most challenging aspects of the hot-swapping mechanism is establishing the quiescent state before the state transfer can occur. The process is explained in more detail in this section. A more complete description of the implementation is available in [?].

Upon swap initiation, the indirection pointer is modified to point to an interposing object called the *mediator*. This mediator object is a generic object capable of handling the swapping of any component, regardless of the interface it exports. This object mediates calls from the time the swap has been initiated, to when the swap has completed. Depending on the state of the swapping operation, the mediator will either forward the call immediately to the original component, suspend the thread associated with the incoming call, or forward the call to the new component.

There are three phases associated with the swapping operation: *Forward*, *Block*, and *Completed*. During the *Forward* phase, the mediator tracks new incoming calls by their thread identifiers and increments an in-flight call counter. It decrements the counter when these invocations return. The mediator stores the thread identifiers in a hash table so that recursive component invocations by the same thread can be identified and allowed to continue even during the next phase. This is required to prevent deadlock, which would occur if we suspended a component-recursing thread. The hash table is also used to save register values used for transparent call forwarding and call returning. The *Forward* phase continues until we have gained knowledge of all in-flight calls to the object; that is, there are no more in-flight requests that were started prior to the swap initiation.

We can determine that there are no more in-flight requests by using the K42 thread lifetime tracking mechanism. In K42, a generation number is associated with each thread to indicate the moment when the thread was activated. The current thread generation is incremented when all the threads activated prior to the current generation have completed [?]. Observing generations passing away allows us to determine when all the threads activated prior to swap initiation have finished, and so all pre-swap in-flight requests have completed.

During the *Block* phase, new incoming calls are first checked to see if they belong to one of the in-flight threads tracked by the hash table. If so, it is a recursive component invocation and is forwarded to the original component. Otherwise, the thread is a new incoming thread, and it is suspended by the mediator. Once the call count reaches zero, there are no more threads executing within the original component and the mediator has established a quiescent state. At this point, state transfer is performed so that subsequent requests to the new component are serviced using the most recent state of the original component. The *Block* phase may seem to add undue delay to the responsiveness of the component. However, in practice the delay depends only on the number of tracked calls which are generally short and few in number.

In the final phase, called the *Completed* phase, the mediator removes its interception to the indirection pointer, and future calls will be directly handled by the new component. All the threads that were suspended during the *Block* phase are resumed and these calls are forwarded to the new component.

Our approach to swapping live components separates the complexity of swap-time in-flight call

tracking and deadlock avoidance from the implementation of the component itself. Transparent call interception and mediation are facilitated by the component system infrastructure. The mediator is used only during the swapping process and hence adds no overhead when not swapping. Structured as a locality-optimized concurrent object, the mediator has good multiprocessor performance and scalability. Besides the component state transfer, the rest of the swapping process is automated by the swapping mechanism, allowing for easy addition of components that wish to take advantage of the hot-swapping capability.

4 Preliminary Results

To gain some initial insight into the performance of hot-swapping components, we measured some of the base costs and experimented with components which are optimized for different multiprocessor workloads. We present results for both a toy component and a more substantial component from the K42 memory management subsystem. The results we present in this section were gathered on an IBM S70 Enterprise Server with 12 PowerPC RS64 processors clocked at 125.9 MHz and a 4MB unified L2 cache.

The mediation overhead associated with forwarding an object method invocation to the original object prior to state transfer is about 633 instructions. This involves: (*i*) the register saves and restores, (*ii*) the mediator prolog (phase check and hash table insert), and (*iii*) the mediator epilog (the hash table retrieve and delete). While this overhead is non-negligible, the cost is incurred only for those method invocations that take place during the *Forward* phase. After the *Block* phase, the invocations that are redirected to the new object do not perform hash table operations nor do they execute epilog code. The delay associated with a swap on an idle uniprocessor with no active threads and using an optimized state transfer is 2786 instructions. On a multiprocessor, this cost occurs in parallel. This delay will grow with the number of in-flight calls to the object being swapped. However, due to our use of thread tracking, the delay will depend on the number of threads in the address space, and not just those executing calls to the object.[3]

Our toy component is a simple counter.[4] We explore two implementations: one optimized for concurrent updates and the other for concurrent reads, referred to, respectively, as the 'Partitioned Counter' and the 'Shared Counter'. Figure **??** illustrates the performance of these two counters under a two-phase multi-threaded workload. Each thread first performs 100000 updates, enters a barrier, and then performs 100000 reads. Each data point represents one independent run in which the counter was accessed concurrently by one thread per processor. The x-axis indicates the number of processors for the run and the y-axis (note the log scale) is the average runtime for a thread divided by 100000. As can be seen, the overall performance of both 'Partitioned' and 'Shared' counters degrade rapidly as the number of processors increase since their worst case behaviour dominates. However, if we start the experiment with the 'Partitioned Counter' and then after the second phase begins we swap it for the 'Shared Counter', we obtain performance which reflects the benefit of adapting the counter to the phase of the workload.

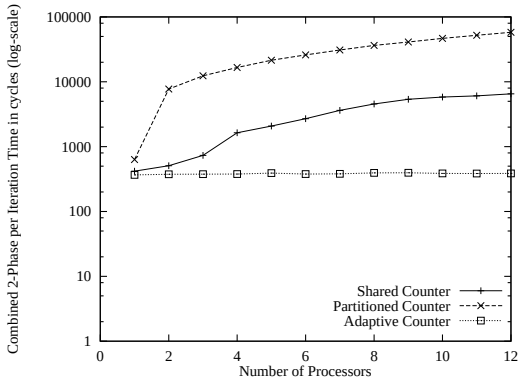

Figure 1: *Performance of different counter object implementations*

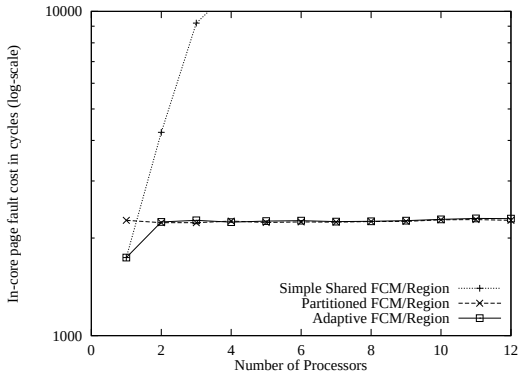

Figure 2: *FCM/Region performance comparisons*

[3] Again, we assume an event driven system in which safe points are frequently reached.

[4] Typical components in K42 are of much larger grain.

Figure ?? illustrates a simple micro-benchmark which measures the average in-core page fault cost in K42 for a multi-threaded program across a range of system sizes. Here, each thread of the program is accessing an independent portion of a shared region of its address space. There are two fundamental components involved in the in-core page fault path: the Region and the File Cache Manager (FCM). In the experiment we test three different cases. In the first, we use the 'Simple Shared FCM and Region' version of the components and observe that they perform very well on one processor but poorly on two or more processors (going off our scale in the figure). In the second case, we use the 'Partitioned FCM and Region' components and see that their performance scales well with increased system size but with a 30% higher base uniprocessor cost. This is illustrative of the uniprocessor performance cost which is common for the more complex data structures and algorithms necessary to support high levels of concurrency. Motivated by the previous two cases, the third case starts with the 'Simple Shared FCM and Region' components but after a fixed number of iterations under high contention, we initiate a swap to 'Partitioned FCM and Region' components. In this test case, we see that the swapping facility, even with 12 concurrent threads inducing contention on the components being swapped, has minimal impact on the overall performance. This supports our contention that hot-swapping is a promising mechanism for making performance sensitive changes to adapt to variations in workload.

5 Conclusion

Although there is a large body of prior work focusing on the downloading and dynamic binding of new components, there has been surprisingly little work on swapping components in a live system while they are in use. Hjálmtÿsson and Gray describe a mechanism for updating C++ objects in a running program [?], but their client objects need to be able to recover from broken bindings due to an object swap and retry the operation, so their mechanism is not transparent to client objects. Pu et al. describe a "replugging mechanism" for incremental and optimistic specialization [?], but they assume there can be at most one thread executing in a swappable module at a time. Our method for reaching a quiescent state is similar to the approach taken by McKenney and Slingwine [?].

We have described our mechanism for supporting hot-swappable components that is totally transparent to the clients of the component and that (in our system) adds zero overhead when a component is not in the process of being swapped. The results of our preliminary experiments show that the idea and concept is promising and warrants further investigation. We see significant advantages in being able to swap components in a live system (especially in our case for improving performance in multiprocessor operating systems and servers), and we are in the process of developing standard, generic tracing and debugging (interposition) objects that can be swapped in and out as needed. The limitation of our mechanism is that it assumes an event-driven (service-oriented) system where threads are relatively short-lived (and hence is not applicable to traditional applications), but we argue that many server-based systems, including operating system kernels, system servers, database systems, and network servers have this characteristic.

References

[1] M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm. Customization Lite. In Proc. 6th Workshop on Hot Topics in Operating Systems, 1997.

[2] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Symp. on Operating Systems Design and Implementation*, 1999.

[3] G. Hjálmtÿsson and R. Gray. Dynamic C++ Classes: A lightweight mechanism to update code in a running program. In *Proc. USENIX Annual Technical Conference*, 1998.

[4] Kevin Hui. The Design and Implementation of K42's Dynamic Clustered Object Switching. M.Sc. thesis, Dept. of Computer Science, University of Toronto, 2000.

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

[6] Paul E. McKenney and John D. Slingwine. Read-Copy Update: Using Execution History to Implement Low-Overhead Solutions to Concurrency Problems. In *Parallel and Distributed Computing Conference*, 1998.

[7] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (SOSP 1995).