# Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly

Andrew Baumann* Jonathan Appavoo† Robert W. Wisniewski†
Dilma Da Silva† Orran Krieger† Gernot Heiser*

* *University of New South Wales and National ICT Australia*
† *IBM T.J. Watson Research Center*

## Abstract

Patches to modern operating systems, including bug fixes and security updates, and the reboots and downtime they require, cause tremendous problems for system users and administrators. Dynamic update allows an operating system to be patched without the need for a reboot or other service interruption. We have taken the approach of building dynamic update functionality directly into an existing operating system, K42.

To determine the applicability of our update system, and to investigate the changes that are made to OS code, we analysed K42's revision history. The analysis showed that our original system could only support half of the desired changes to K42. The main problem preventing more changes from being converted to dynamic updates was our system's inability to update interfaces. Other studies, as well as our own investigations, have shown that change to interfaces is also prevalent in systems such as Linux. Thus, it is apparent that a dynamic update mechanism needs to handle interface changes to be widely applicable.

In this paper, we describe how to support interface changes in a modular dynamic update system. With this improvement, approximately 79% of past performance and bug fix changes to K42 could be converted to dynamic updates, and we expect the proportion would be even higher if the fixes were being developed for dynamic update. Measurements of our system show that the runtime overhead is very low, and the time to apply updates is acceptable.

This paper makes the following contributions. We present a mechanism to handle interface changes for dynamic updates to an operating system. For performance-sensitive updates, we show how to apply changes lazily. We discuss lessons learned, including how an operating system can be structured to better support dynamic update. We also describe how our approach extends to other systems such as Linux, that although structured modularly, are not strictly object-oriented like K42.

## 1 Introduction

Patches and updates to modern operating systems are a significant problem for users and administrators. Operating system vendors are releasing an increasing volume of patches at a higher frequency [11], and these patches require restarting services or rebooting the whole system, resulting in downtime that is becoming increasingly costly. This downtime, even if scheduled, is expensive, causing administrators to trade-off its cost against the risks of remaining unpatched. Many do not apply well-announced and widely-propagated security-critical patches for weeks [23]. Furthermore, rebooting a system causes loss of transient state, and thus may be a serious inconvenience to its users.

When we discuss updates, we are considering primarily the kinds of changes that are released in the maintenance process of a mainstream operating system after a major release. For example, bug fixes, security fixes, and performance improvements. Significant new features are rarely released in this way, because application software would need to be updated to take advantage of them. These updates are usually released regularly, depending on their urgency, and are developed and tested by the operating system vendor before distribution to system administrators.

### 1.1 Existing approaches

Here we give an overview of current approaches to updating systems without loss of service; more closely related work is discussed later, in Section 8.

Traditionally, the solution to the problems of updates and reliability has been to use redundant hardware; that is, either specialised hardware that processes requests on identical machines, or more commonly commodity hardware. If commodity hardware is used, and the service maintains state (unlike a traditional web server), software support must be provided to maintain synchroni-

sation between the redundant systems.

A similar approach is to use virtualisation instead of physically separate hardware. This also requires software support to maintain synchronisation, or a mechanism to migrate applications between virtual machines at update time [22].

Outside the operating system, the most common approach to dynamic updating is to build-in support for updates in a language-specific [3, 10, 14, 20, 25] or domain-specific [1,6] manner. Our work can be seen as a domain-specific approach to achieving dynamic update for operating systems.

## 1.2 Our approach

Our goal is to provide dynamic update support within the operating system itself, without the need to re-implement or significantly restructure its code. Ideally, it should be possible to load an update into the system similarly to the way we can load kernel modules to add functionality.

We observe that pressures of software development, safety, extensibility, and configurability are driving modern operating systems, even those with a monolithic kernel structure, to become highly modular or componentised. For example, several projects have added more modularity to Linux to enable fault isolation [26, 28]. Software construction techniques such as abstract types, data hiding and encapsulation, and separation of concerns are features of these modular interfaces.

Our approach is to leverage module boundaries to update the code and data within a module without affecting the rest of the system. We provide mechanisms for safely updating a specific module, and transforming the data structures maintained by that module. We repeatedly use those mechanisms to update all modules involved in a larger change, achieving a whole-system update as a series of small self-contained changes. Despite the module-based approach, we are able to apply updates even when a module's interface changes.

## 1.3 Overview

In previous work [4], we developed a prototype implementation of dynamic update for the K42 research operating system, and tested it with a small number of hand-picked update examples. This prototype is outlined in the following Section 2.

Selecting a small number of changes to convert to dynamic updates, as we had done previously, showed that our system worked but did not help to determine whether the system would be able to apply all of the relevant changes as dynamic updates. To properly assess the coverage of our dynamic update system, we conducted a study of the K42 revision history, that is described in Section 3. In our previous prototype, changes to module interfaces could not be applied as dynamic updates. We had expected such changes to be rare, however the results of our study showed that they were relatively frequent—the majority of new features and a significant proportion of bug fixes and performance improvements included changes to interfaces. Our own investigation and other studies have found that the same is also true for Linux.

To provide a more complete update system, we designed a dynamic update mechanism for an OS that handles interfaces changes, as described in Section 4. This section also describes the new lazy update functionality we have developed for mitigating the performance impact of large updates. Revisiting the revision history analysis with these improvements, we found that the majority of all changes, and an even higher proportion of maintenance changes could now be converted to dynamic updates.

We have measured the performance impact of adding the capability to perform dynamic updates to the system, and found it to be negligible. We have also measured the costs incurred when a dynamic update is applied. These results are presented in Section 5, along with a description of example updates enabled by the added functionality. We then discuss lessons learned from implementing dynamic update in K42 in Section 6, including how we would structure the system differently to better support it.

The work presented here enables dynamic updates to the operating system, but also raises some questions for future research. In Section 7 we discuss open issues in the area, and in Section 8 we describe related and complementary work. Section 9 concludes.

To summarise, our primary contributions over the previous work are as follows. We have conducted a broad analysis of changes in the revision history of an OS, and used this to assess the applicability of our update system. Based on the limitations identified, we have extended the model to support interface changes, which significantly increased the scope of changes that we could support as dynamic updates. We have also added support for lazy conversion of data structures, because the performance impact of converting all data structures at once could be dramatic. Finally, we include an evaluation of the performance characteristics of our system, and a discussion of our experiences using and developing it.

# 2 Background

We previously developed a prototype dynamic update system [4, 5]. Because that work is essential background information, we briefly summarise it here.

## 2.1 Design

We identified several fundamental requirements for an operating system to provide dynamic update capability. The most important of these are a modular system structure, a mechanism for detecting a safe point to update a given module, and state tracking and transfer mechanisms to locate and transform the state information maintained by a module.

Given these requirements, the generic update process for a single module is as follows: First, the code associated with an update is loaded into the system by a kernel module loader, or similar mechanism. Next, a *state tracking mechanism* is used to locate all data instances affected by an update. Then, using a level of indirection on module invocations, we block any new accesses to the affected module. Once the *safe point mechanism* detects that the module is idle, or *quiescent*, we update the code in the module and transform its data structures using the *state transfer mechanism*. Finally, having finished the update, the new module is made accessible, and any blocked calls are resumed.

## 2.2 Dynamic update in K42

K42 is an operating system project targeting scalability and customisability [13]. It runs primarily on 64-bit PowerPC systems, and supports the Linux API and ABI. K42 is implemented in C++, and is object-oriented: each resource managed by the kernel is provided by one or more distinct object instances. To improve scalability in an SMP system, all objects are accessed indirectly through a global *object translation table* (OTT); this indirection also enables dynamic update.

In K42, *state-transfer functions* are implemented for each object, and convert an object's internal state to, or from, a common intermediate representation. Our system detects *quiescence* by tracking the lifetime of kernel threads, and providing a mechanism to determine when all threads that were active when access to an object was blocked have terminated. This works well, because the kernel is event-driven, and its threads are short-lived.

Our original implementation added two features to K42 to support dynamic update: a kernel module loader and a factory mechanism. The module loader is similar to the one used in Linux, but simpler because updated code is only accessed indirectly through object references. The factory mechanism is responsible for the creation, destruction, and tracking of object instances within K42 via the factory design pattern [9]. Factories in K42 are live objects accessed through well-known references, one per class. They allow us to update all the objects affected by a code change, ensure that future instantiations use the updated code, and track when all the objects of a given class have been updated.

Using these foundations, we were able to apply dynamic updates to K42. We hand-picked some interesting changes from the K42 revision history, converted those to dynamic updates, and applied them to the running system.

# 3 Analysis of CVS history

In this section, we describe a study we have performed of changes from the K42 CVS revision history. Questions we sought to answer included:

- What change types are seen in K42's development?
- What proportion of these changes are bugfixes, security fixes, or performance improvements that would be shipped in maintenance releases?
- How many, and what kind of changes could we apply using our dynamic update mechanism?

The broader question of how operating system code evolves is also not well understood, and is a rich area for further investigation.

## 3.1 Method

K42 was developed over a period of nine years (the first revision is from March 1997), by around five to ten developers. Hence, there is a lot of revision data in the repository to be examined: 4,814 files and 56,199 revisions in the core modules we examined.[1]

One of the drawbacks of CVS is that it operates only at a file and revision level, and does not track any dependencies between files or directories. Thus, we first had to develop mechanisms and heuristics for extracting independent transactions or changes from CVS revisions. This required two assumptions.

First, we assumed that each commit operation by a developer was a single logical change or feature. This is usually true, but not always. A few developers tended to commit unrelated changes together. This means that we see fewer and larger changes than we should, so our results are pessimistic.

Second, we assumed that after each commit the repository was in a consistent state. That is, it could be expected to compile and run correctly. Obviously developers make mistakes, so this is not always true. However, K42 includes an extensive set of regression tests that developers usually run before committing, so in the majority of cases the assumption was valid.

Given these assumptions, we developed a modified version of the *slurp* tool [16] to process the CVS repository data and import it into a database for further analysis, and used an algorithm described by Zimmermann

and Weißgerber [30] to group related CVS revisions into logical transactions. For each source file revision, we also filtered out all the comments, reformatted the code in a consistent style using an *indent* tool, and computed the differences between the cleaned and reformatted source. This significantly reduced the number of irrelevant changes that needed to be examined.

Using this data, we performed two types of analysis. First, we used automatically-computed contextual information to determine which transactions changed only code inside dynamically-updatable objects and could therefore be developed into dynamic updates. Second, we randomly selected transactions from our sample and manually inspected them to gather more accurate and more detailed information about what types of changes are possible, and specifically what prevents changes from being converted to dynamic updates. Based on that result, we estimated from the overall set of changes what proportion could be converted to dynamic updates.

For both analyses we considered only transactions that altered some kernel code. Specifically, the source differences computed in the final step were non-empty, and at least one of the files modified by the transaction was within the `os/kernel` directory. Apart from some common library code, this directory contains the K42 kernel, including process and memory management, IPC mechanisms, exception handlers, boot code, and Linux glue code. File-systems and device drivers are reused from Linux, and their source is maintained elsewhere.

## 3.2 Automatic analysis

Most of K42 consists of objects accessed indirectly through the object translation table. However, some parts, such as the exception handlers and parts of the scheduling code, are not accessed indirectly, and therefore are not dynamically updatable. To calculate what proportion of changes could be converted to dynamic updates, it is necessary to determine which changes affect only code inside these dynamically-updatable objects.

We would have liked to determine what functions, data structures, and objects were changed by each transaction in the repository. This implies parsing the code. However, a normal C++ parser would read all the header files included by a particular file; effectively it would require reconstructing the K42 source tree for every transaction, which would be very slow. To avoid this, we wrote a pseudo-parser handling just enough of the language (for example, the *class* keyword, function definitions, and braces) to identify a *program context* for every line of C++ source. A program context is a function or class name, or a special *global* context (used, for example, for preprocessor directives).

Given this contextual information, we identified for each transaction any classes or functions added or deleted by that transaction, and also any that were modified. We then examined every transaction that included a change to kernel code (a total of 3618 transactions), and categorised them based on those that added contexts, those that deleted them, and those that just modified code within existing contexts.

Using a list of classes known to be dynamically updatable, we then identified the transactions that changed only dynamically-updatable code. Our list included some classes that do not yet have state-transfer functions or factories, so are currently not updatable. We included these, because the addition of state-transfer functions and factories is relatively simple (the changes are confined to the class itself), and because we believe that showing the limitations of the model is more meaningful than showing those of the K42 implementation.

We found that 22% of transactions only modified or added methods in dynamically-updatable classes. A number of common problems prevented more transactions from being classified as dynamically-updatable:

- changes to code for testing, tracing, or debugging, that would not be released as updates, and so are irrelevant to our target problem;
- changes to initialisation code that would instead be implemented as part of the state transformation and dynamic update load process;
- changes to simple classes that aren't themselves dynamically updatable, but are encapsulated within dynamically-updatable objects, and so could be updated as part of the surrounding object;
- changes to the global context, such as preprocessor definitions or global declarations, that would be handled differently for a dynamic update.

If we include these, the result rises to 48%. If we exclude changes before 2002, when K42 was in a more developmental phase, the total proportion of dynamically updatable transactions rises to 55%.

Due to the automatic nature of this analysis, these results include a certain amount of noise. For example, some changes were committed to the tree, then reverted because they caused regressions, and later committed again with fixes; these should be counted as only one transaction. Other transactions included, upon closer inspection, multiple independent changes. Many changes performed cleanup actions, such as moving code between header files, splitting or merging classes, and so on.

These examples show the limitations of automatically analysing the revision history. The result is skewed by a large number of changes that would never need to be

developed into dynamic updates, however it gives us a reasonable lower bound for the proportion of dynamically updatable changes. For a more accurate result, we conducted the manual analysis described in the following section.

## 3.3 Manual sampling

The automatic analysis gave us useful information about the overall proportion of dynamically updatable changes, but these results include all the changes in K42's revision history, and the revision history of an experimental operating system does not mirror what would happen in the maintenance of a released operating system. Changes that happened frequently in K42, such as new features, code cleanups or debugging changes, would not be shipped in maintenance updates. For a more accurate analysis of the applicability of our dynamic update system to the specific types of change in which we are interested, we conducted a manual investigation using a sample of the CVS transactions.

We developed a simple web application allowing a human analyst to examine randomly-selected transactions. For each transaction, the analyst was shown the commit log message and other meta data, the source code differences for affected files, and the list of changed program contexts computed by the previous automatic analysis. The analyst then assigned each transaction to a number of categories, using their knowledge of the K42 code, as well as an understanding of what could be changed by a dynamic update. Our goal was to examine a sufficiently large number of transactions to obtain statistically significant conclusions about the proportion of dynamically updatable changes. In total, we have manually analysed 250 transactions.

Some transactions were considered irrelevant to the analysis, and ignored. These included a change that was reverted and then recommitted later, a small number of transactions that included many unrelated changes, and many changes that were functionally-equivalent such as a reorganisation of header files, changes that only added debugging output, changes to preprocessor directives, and so on. In total, 39% of the transactions that we examined were ignored.

We then looked at the change as a whole, and placed it into one of five categories based on its main purpose: bug fixes, security fixes, minor/maintenance performance improvements, new features, and changes for non-functionally-equivalent cleanup or restructure. Of the non-ignored transactions, 48% were restructuring, 36% added new features, 11% were bug fixes, and the remaining 5% performance improvements. We found no security fixes. Because K42 has been used to date only for research purposes, security holes that would necessitate fixes have not been uncovered. However, because security fixes are a subclass of bug fixes, and tend to be of a small, isolated, and feature-less nature [2], we expect that results for the bug fix category will be a good indicator of our system's support for security updates.

We also examined the code differences and determined what was affected by the change: data structures, interfaces, multiple objects, and library functions (recall that we selected any changes affecting the kernel source code, which could also include changes to user libraries).

Finally, we decided whether the change was convertible to a dynamic update. Of the transactions categorised as bug fixes or performance improvements, which we will refer to as maintenance changes, only 50% could be converted to dynamic updates. Of the non-updatable maintenance changes, 58% were ruled out because they changed interfaces, and the remainder changed non-updatable exception handler code. In the other categories, only 11% of new features and 6% of code restructures could be converted to dynamic updates. These results are shown as the *simple update* case in Figure 2.

## 3.4 Conclusions

Extending our results from a case-study analysis of the K42 revision history to more general conclusions about the applicability of our dynamic update model is potentially error-prone. The revisions in the main branch of a research operating system do not necessarily reflect the maintenance and update release process of a production system. Nevertheless, our study gives an indication of the updates we can expect to see in systems code. In a production operating system in maintenance mode, we would expect far fewer broad restructures and added features, and a greater proportion of performance and security updates or bug fixes.

We were surprised by the high incidence of changes to interfaces, even among the maintenance updates. An interface change in K42 is any change to the virtual methods defined for a class, that would cause code compiled against the previous definition to behave incorrectly. This includes the addition or deletion of methods, arguments, or changes to types, none of which were supported for dynamic update. We were aware that this was a limitation, but believed that it was not significant, as most updates would not change interfaces. However, our results showed that a surprisingly high proportion of kernel updates did require changes to interfaces.

To verify that this problem was not unique to K42, we inspected recent stable releases of the Linux kernel: versions from 2.6.18.1 to 2.6.18.6 inclusive and version 2.6.19.1. These releases include relatively few changes, the largest uncompressed patch being 250 kilobytes in size, and contain only bug and security fixes. However,

four of the seven versions examined included changes to the prototypes of non-inline kernel functions, confirming the prevalence of interface change in Linux.

Other researchers have reached similar conclusions regarding the need to support interface changes. Neamtiu *et al.* [17] conducted a study of source code evolution in several common open-source programs, including the Linux kernel, with the goal of informing the design of dynamic update systems. They concluded that changes to type definitions and function prototypes were both common enough to be an important feature for a dynamic update system to support. Furthermore, another recent study of collateral evolution in Linux device drivers [21] highlighted the problems associated with changes to interfaces in that system.

Therefore, it is clear that for our dynamic update system to be usable, it must support evolution of interfaces within the kernel. In the following section we will discuss how to address this challenge.

## 4 Extending to complex dynamic updates

Based on our experiences and results from the previous section, to enable more updates to be applied, and to increase the applicability of our system, we have made a series of improvements to its design and implementation. Here we discuss the most significant: support for interface changes and lazy update. We also describe the process of developing and applying an update.

### 4.1 Interface changes

Our previous design and implementation did not support changes to object interfaces. As shown in the CVS analysis, this was a serious limitation on the applicability of our system. When an object's interface changes, any calling objects that depend on the interface must also be changed. The obvious solution is to update all affected objects in a single atomic operation, however blocking and updating multiple objects may be unworkable. For complex changes, it effectively requires quiescence across the entire kernel, leading to large delays, and potential deadlock and correctness issues (for example, missed interrupts could cause the system to lockup or crash).

From a closer examination of the changes to interfaces observed in the K42 revision history, we found that most of the changes were relatively minor. These included: adding or renaming functions; removing parameters from functions; and extending the parameter list of existing functions, but providing a default parameter to avoid updating all the existing call points.

Informed by these observations, we decided to use the object adaptor design pattern [9]. Adaptors wrap a class

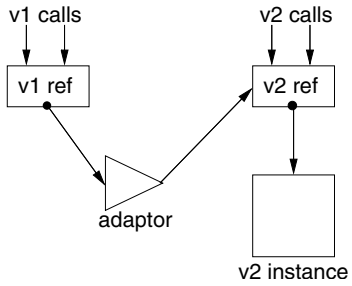


Figure 1: Adaptor object

to make it provide a different interface. In K42, adaptors were implemented for dynamic update, and operate transparently to other objects through the use of the object translation table. They can maintain their own state information, and are able to intercept and rewrite all function calls from old un-updated callers of the object. Changes made possible by adaptors include:

- adjusting virtual method numbers, when functions have been added;
- shuffling parameter registers, and computing or supplying defaults for new parameters;
- altering return values, or directly returning a value (such as an error code) without calling the object.

Not all interface changes can be expressed by an adaptor. In particular, changes that are not backwards-compatible, or where the old interface cannot be provided by operations in the adaptor or on the updated object, are not possible. This includes changes where functionality is removed, or complex restructuring changes, such as when an object's functionality is split into several other objects. However, the forms of interface change supported by adaptors are sufficient for maintenance changes, as will be shown in Section 4.4.

Our design is shown in Figure 1. Any change that alters an object's interface requires an adaptor to be supplied along with the updated code. When the dynamic update is applied, the new object with the updated interface is installed on a new object reference, and an adaptor object is instantiated for the old reference, forwarding calls to the underlying object. Then, caller objects are progressively updated to directly invoke the new interface. Once all old caller objects are updated, as determined by the relevant factory objects, the old reference and the adaptor object are destroyed.

Some low-level code in K42, such as the initial page-fault handlers, is not part of the object system and therefore not updatable. If any of the objects called by such low-level code are updated with an adaptor, then the adaptor will be required permanently, because it is not otherwise possible to update the calling code to use the

new interface. Fortunately, in K42 there is very little code in this category.

The use of adaptor objects follows our fundamental design principle of applying dynamic updates as a series of small independent changes. Adaptors allow us to update the system progressively, without the need to concurrently block access to multiple objects. Adaptors impose additional overhead on all function calls to an affected object, but this overhead is only transient—as the calling objects are updated to versions that support the new interface, which happens as part of the overall update, the adaptors are removed.

## 4.2 Lazy update

Our original model transformed every object at the time an update was loaded; other dynamic update systems that support changes to data structures have also taken this approach [8, 19]. However, this presents a scalability and performance problem, because some objects may have thousands or more instances present within the kernel; for example, the objects associated with open files or memory regions. When testing a K42 update that altered the in-memory data structures of each open file on a loaded system, we found that the system performance was severely degraded while converting all the affected objects.

To illustrate the scale of this problem, we used the `/proc/slabinfo` file to count instances of different kernel data structures on a moderately-loaded file and compute server running Linux 2.6.18. We found 1.9 million each of the filesystem's *inode* and *vnode* structures, 234,264 blocks in the buffer cache, 51,301 virtual memory areas, and 14,437 open files, to take a few examples. If any of these data structures were changed, it would not be feasible to delay the system's execution while they were all updated.

To address this problem, we implemented the ability to perform the dynamic update lazily [6]. When a lazy update is loaded, affected object references are changed to point to a special lazy-update object. The first time this object is invoked, it initiates the actual update, restarts the method call that triggered it, and then removes itself from the affected object reference. Laziness mitigates the performance impact of updates involving many objects by spreading out the load, because rather than transforming all object instances at once, objects are gradually converted as they are accessed. It achieves this while still guaranteeing that the old code will not be invoked once the initial process of installing the lazy-update objects is complete.

Lazy update also allows us to avoid unnecessarily converting objects that are not invoked between updates or ever again. If an object has been only lazily updated, and another update to that object is loaded, we could use the state-transfer functions from both update versions in sequence, avoiding the cost of twice achieving quiescence in that object. Another modification of the technique would be to combine lazy update with a daemon thread that runs at low priority, updating objects as the system's idle time allows.

## 4.3 Update process

The changed process for developing and applying a dynamic update, as opposed to the simpler version outlined in Section 2, is as follows. To build a dynamic update, we take the new version of any changed classes, develop and add necessary state-transfer functions, and compile them together with code that initiates the update to form a loadable module. If the update changes a class interface, then an adaptor object must also be implemented and included in the module.

The update module is then loaded into the kernel. Its initialisation code triggers an update of the factories for the affected classes. Next, a new factory walks through the old object instances, either initiating a direct update, or marking them to be updated lazily. Presently, the developer of an update determines whether to use laziness, but this could also be implemented by heuristics in the factories; for example, if there are more than 100 live instances, use lazy update.

To update an individual object, if an adaptor is being used, it is first installed on the old reference, then the state is transferred to the new object while both are quiescent. If an adaptor is not required, the object's reference does not need to change, because all calls conform to the same interface and thus can be intermingled—in this case, the new object simply takes over the old reference.

When an object is updated to understand altered interfaces, its state-transfer function must locate the new reference for any objects whose interfaces have changed. This is done using a special function implemented in the base classes for all objects, that returns the canonical reference for a given object.

As object updates complete, either directly or lazily, the old objects and lazy-update objects are destroyed. Finally, when all old objects of a given type are updated, as determined by the relevant factory, two cleanup operations occur. First, any adaptor objects can be removed and their references reclaimed. Second, the code used for the loaded module corresponding to the previous version of the class, and other static kernel memory associated with that class is reclaimed; however, our module loader implementation does not yet free memory.

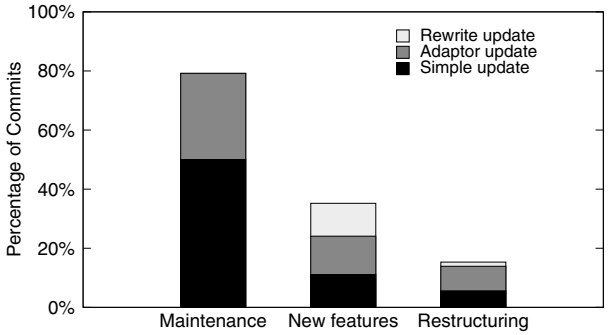


Figure 2: Results of manual CVS analysis

## 4.4 CVS analysis revisited

With support for limited interface changes in the form of adaptors, we revisited the manual CVS analysis of Section 3. We also considered whether the changes could have been altered slightly to allow them to be dynamically updated. For each transaction, there were now four possibilities: the change could be updated without adaptors, the change could be updated only with an adaptor, the change could be updated only after some simple rewriting of the code (possibly also with an adaptor), or the change could not easily be updated. These results are shown in Figure 2.

We found that all of the maintenance changes that altered interfaces could have been supported through the use of an adaptor, raising the total of updatable maintenance changes to 79% (none required rewriting). Of the other categories, 35% of new features and 15% of code restructures could now be converted to dynamic updates. All of the complex interface changes that could still not be supported by the use of an adaptor were found in the new feature or code restructure categories, confirming that the limited form of interface change supported by adaptors is sufficient for maintenance purposes.

In the course of analysis, we found it common for transactions that were otherwise dynamically updatable to include minor related changes to add test code, alter initialisation functions, or perform some other cleanup that was not updatable, or was part of another object. These other changes would have been ignored in developing the dynamic update, so we did the same in our analysis and added another series of flags to note when this was done. Of all the dynamically updatable changes, 39% fell into one of these categories. Of only the maintenance updates, we ignored minor parts of 33% of the changes.

The new results in our analysis show that approximately 79% of maintenance changes could directly be converted to dynamic updates. We regard this as a worst-case for our model, because the changes were developed without considering dynamic update, and because some changes to exception handlers might instead have been implemented at a higher level in dynamically-updatable code. We expect that in the maintenance of a real system it would be possible to develop most of the remaining changes as dynamic updates. We will discuss this further in Section 6.2.

# 5 Evaluation

We conducted experiments to measure the overhead of our update mechanism, and the performance of our system when applying various updates. The results of this evaluation are reported in this section, along with a description of more complex updates enabled by our improvements.

In several of the experiments reported below, we used the ReAIM implementation of the AIM7 multi-user benchmark, in the *alltests* configuration. This benchmark exercises OS services such as IPC mechanisms, file IO, signal delivery, and networking. It was modified slightly to work with K42: we replaced the test using Unix-domain sockets with UDP sockets, altered some code to handle different error return values from K42's Linux emulation library, and prevented the benchmark from removing shared memory regions at the end of its run, because this is not yet supported by K42. We ran the benchmark inside a RAM disk, to avoid IO latencies not imposed by the OS.

All experiments reported here were conducted on an Apple Xserve system, with two 2GHz G5 processors and 512MB of main memory. We built K42 in the no-debug configuration, and ran it in dual-processor mode.

## 5.1 Costs of mechanism

In this section we examine the added runtime costs of having support for dynamic update in the system. This is much more important than the time to apply an update (which we measure in Section 5.2) because we expect updates to be infrequent events, and because even if the system experiences a slowdown while an update is applied, the advantages over rebooting are significant.

### Indirection overhead

First, we consider a property that is part of the fundamental structure of K42: the object translation table's additional indirection on object calls. This indirection adds extra overhead to each object invocation; an object call in K42 requires 6 instructions, instead of 5 for a regular virtual function call. The added instruction is a dependent load, however, because object references are allocated sequentially from a single region of memory, the

object translation table is dense, and thus the extra load is likely to be cached.

It is not possible to directly measure the cost of object indirection, because it is a fundamental part of K42's structure. Instead, we estimated the overhead of object indirection by instrumenting the object system to count the number of indirect object invocations during a run of the ReAIM benchmark. Multiplying this by the number of cycles required for a load from the second-level cache, we estimated the overhead of indirection at less than 0.1% of the total running time on the unmodified system.

Arguably a bigger impact comes from not having a static system structure, preventing application of such cross-module optimisations as inlining, path straightening and dead-code elimination. Furthermore, such a structure replaces direct with indirect branches, and affects the performance of hardware branch prediction. However, any system offering basic module-loader functionality suffers the same problem, and kernel module loaders are now common in commodity operating systems, so this cost has been accepted in those systems.

#### State tracking overhead

The other added overhead comes from factory objects. During object creation, a factory is now involved, and records the new object's reference. During deletion, the reference is removed from the factory's data structures.

To measure the impact of factories on system performance, we used the ReAIM benchmark described previously. We implemented factories for process objects, one of which exists for each process, and the core memory management objects,[2] a pair of which are created for each open file or mapped memory region. These represent a significant proportion of the objects created in the kernel during benchmark activity: 1,791,808 of all 2,996,920 objects created during a ReAIM run, or 60%, were created using a factory. Hence, we would expect any impact to be visible.

We benchmarked the unmodified system and the modified system with factories added for the above objects. We repeated each experiment three times, taking the average of the peak jobs-per-minute result from each run. We measured a slight (less than 0.5%) performance improvement with the factories present. This is well within the noise caused by changes in code layout and cache behaviour, so we concluded that the use of factories within the kernel has negligible performance impact on the overall system.

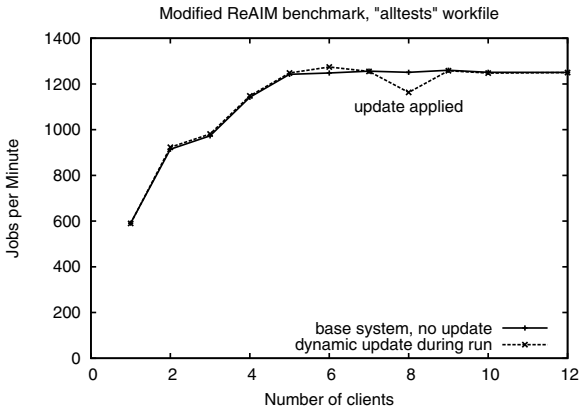


Figure 3: *sync* update applied while benchmarking

## 5.2 Experiences

Here we describe several example updates enabled by the functionality introduced in Section 4. We also present performance results of applying one of those updates.

#### File sync patch

When a file is closed, or when its last mapping is removed, the kernel initiates a *sync* operation. This is known as an *unforced sync*, as opposed to the operation that occurs when a program explicitly invokes the *sync* or *fsync* system calls. Because processes block on forced syncs, but unforced syncs only delay the destruction of some buffers, the implementation was changed to prioritise forced syncs over unforced sync. This involved significant changes to the structure of the kernel's *FCMFile* object, with queues of waiting threads and IO operations now maintained differently.

We converted this patch to a dynamic update, by implementing state-transfer functions for *FCMFile* that restructured the internal queues. In total, 53 lines of state transfer code were written.

Figure 3 shows the result of applying this update during a ReAIM benchmarking run. ReAIM incrementally tests higher numbers of clients, so wall-clock time in this graph runs from left to right. We initiated the update when there were eight clients active, and during the next second the overall system throughput dropped, while 170 instances of the *FCMFile* object had their data structures converted on access. Once the benchmark reached the run with nine clients, all affected instances had been transformed, and the update was complete. This can be seen on the graph as a dip in throughput.

While this update was applied, we used a hardware cycle counter to measure the time required for individual phases of the process. Once the update process was started, it took 7ms to instantiate an updated factory ob-

ject and convert the existing factory, and then 487$\mu$s to mark all the object instances to be lazily updated. At no point was the whole system's execution blocked.

We also used this update to time our module loader. The module loader is unoptimised, but because the dynamic update process starts once the module has been loaded, this is not of concern. It required 241ms to load the update into the kernel and start executing its initialisation code.

#### Use of adaptors in large page change

As part of a change made to improve support for multiple page sizes, K42's *root page manager* (PMRoot) class was modified. The *getFrame* and *freeFrame* methods had an argument added specifying the frame size. In this case, there is an obvious default value that can be used for un-updated callers of the interface: the standard page size of 4 kilobytes. Hence, an adaptor object that modifies calls to the PMRoot object can be used. If the method number of a call matches either the *getFrame* or *freeFrame* methods, the relevant argument register is set to 4096.

As part of the overall update for such a change, once the PMRoot object has been updated and an adaptor installed, it is then possible to update the other objects that call PMRoot. This example shows how, in our model, one logical change is implemented as a series of updates, allowing the system to continue making progress.

#### Interface change to base classes

We provide an example of a dynamic update that is possible with the use of adaptor objects, but has an impact on many objects of different classes due to inheritance. As part of an experiment with new page-allocation policies to avoid fragmentation, a method was added to the *page manager* (PM) class. This class is subclassed by a number of other K42 objects that inherit the new method but also define their own methods. Because a method is added, an adaptor object must be implemented that rewrites the calls made by old callers of the object. Every call made to the old interface with a method number greater than or equal to the newly-added method must have the method number incremented.

The adaptor implementation itself is quite simple, as it only adjusts the virtual function number. However, this example illustrates a scenario where K42's heavy use of C++ and implementation inheritance increases the complexity of updates. Because a method was added to a base class, it was effectively added to all the subclasses, changing their own interfaces, even though the source files did not change. As a result of a single class change, we need to prepare an update for each of the subclasses. A single adaptor implementation suffices, but identifying the affected classes is presently a manual process; we would anticipate a real system automating it, as we will discuss in Section 7.

## 6 Lessons learned

If we had intended K42 to be dynamically updatable from the beginning, there are several ways in which we would have structured it differently. In this section we discuss solutions to some of the problems we encountered adding dynamic update support to K42.

In our manual analysis described in Sections 3.3 and 4.4, we recorded comments noting why a change couldn't be converted to a dynamic update. The problems generally fell into one of the following categories:

1. changes to static code and data structures, such as low-level code in our exception handlers, general debugging services like our GDB stub functions and in-kernel test system, and system initialisation code;

2. specific services that were developed using static structures and enumerations, primarily K42's tracing service, where each trace point has a unique dense trace identifier, but also glue code used to wrap parts of Linux that run in our kernel;

3. very large cross-cutting changes due to fundamental restructuring of code.

While the third category is probably unavoidable in a research system, it would be unlikely for such changes to occur in the maintenance process of a released system. These include mass changes to interfaces that are not backwards compatible, for example, a number of changes were analysed in which a new argument was added throughout deep call chains across multiple objects. This argument cannot be set by an adaptor, and rewriting the code to support invocation either with or without the argument would be very complex and probably introduce bugs. Similarly, if an interface changes such that the functionality formerly provided by one object is now split across two or more objects, it is very difficult to hide this complexity behind an adaptor and still maintain compatibility for un-updated callers of the interface.

There are however some problems from the other two categories for which we have designed solutions.

### 6.1 Restructure of initialisation code

The most common example of a static code problem is the kernel initialisation sequence. This code executes once at boot, consists mainly of calls to class-specific

initialisation functions, and cannot be updated. It makes no sense to update the code itself, as once the system has booted any changes will have no effect. However, an update often does need to include initialisation code, for example when introducing a new class into the system, and in many cases this code is the same as the corresponding boot code.

We envision adding a mechanism that allows programmers to annotate initialisation functions in class header files, and automatically calls those functions in a predictable order at boot time, or when the class is loaded as a dynamic update. This is similar to Linux's *initcall* mechanism [29], which uses annotations on functions to be called at boot.

Another related problem is testing code accessed from the kernel console. Currently this is implemented as static functions, and thus is not dynamically updatable. Although test and debugging code is not important for dynamic update in a production system, we would redesign the K42 test system to allow test functions to be registered dynamically, enabling dynamic update as well as dramatically improving the source.

### 6.2 Exception handlers

Parts of the K42 kernel are not implemented as dynamically-updatable objects, and thus cannot be dynamically updated by our system. This includes low-level exception-handling code, parts of the scheduler, and the implementation of K42's message-passing IPC mechanism. Changes to such code account for the remaining non-updatable maintenance changes in our CVS analysis.

In some cases it is possible to achieve a dynamic update by rewriting other code. For example, on the page-fault path, by implementing a change at a higher level inside the memory management system's dynamically-updatable objects rather than the exception handlers. In other cases, because it is rare for such changes to alter data structures, it may be possible to use indirection available in the exception vectors, or binary rewriting techniques [27] to update the code without the need to achieve quiescence. If data structures were changed, and thus quiescence was required, one could either disable interrupts or run the OS inside a virtual-machine monitor (VMM) [8].

### 6.3 Dynamic tracing support

K42's tracing service generates a binary log from trace points inserted throughout the system, where each trace point has a unique identifier. To simplify the original implementation, and because we had not considered dynamic update at that time, a static enumeration was used to identify and allocate trace point numbers. To support dynamic updates that add or remove trace points, we could change the static enumeration to a dynamic structure, or use a totally dynamic tracing service [7, 27].

### 6.4 Conclusion

In general, increased dynamic update coverage can be achieved by minimising statically-bound code, and wherever possible, using structures created at run-time rather than compile-time. This has other advantages: it makes a system more modular, leading to simpler code, better maintainability, and better extensibility.

Our experience shows that when dynamic update is desired, it is usually possible to modify the relevant system structures to enable it. Dynamic update was an addition to K42 after years of development, so it is encouraging that we were able to add it without major structural changes.

## 7 Future work

### 7.1 Update preparation

The process of preparing updates for K42 is currently manual. Besides changing the code for a class, state-transfer functions and adaptor objects must be implemented, and programmers must determine the dependencies between updates. Although, as we have shown, it is possible to dynamically update a system using our design, the ideal update system would also automate update preparation from source code changes.

This is not a problem that we are addressing, but it is not unique to operating systems. Other work in the dynamic update field has developed tools to ease the construction of dynamic patches [2, 19] and investigated the semi-automatic creation of state transformers [14, 19], and it would be possible to generate common adaptor objects from source code analysis.

### 7.2 Reverting updates

In some cases, an administrator may wish to revert or roll back an update after it has been applied. We expect this to occur rarely, since we are considering maintenance updates that have undergone testing by the vendor before their release. Nevertheless, given the data transfer functions and adaptors (which would be developed as part of the update), a *reversal update* that had the effect of reverting to the previous version could be created. However, in any system where updated code runs in the kernel unprotected, if an update has bugs that cause it to corrupt data structures, recovery may be impossible.

### 7.3 Arbitrary interface changes

As currently designed, our system cannot support interface changes in which the changes cannot be hidden behind an adaptor. While this is not a problem for all but the most substantial new features and code restructuring changes, for completeness we would like to support all dynamic updates.

As explained previously, support for arbitrary interface changes requires blocking all objects affected by a change, and potentially the whole kernel. We would block kernel events at a lower level, such as exception handlers or underlying VMM, before updating the system. Although this precludes servicing requests while the update is applied, it preserves the system's full state, and thus offers significant advantages over rebooting.

### 7.4 Updates outside the kernel

Many OS updates change user-level code such as system libraries. Although we have targeted kernel changes, a complete dynamic update system would also require support for user-level updates. There is nothing preventing our update mechanism from operating at user-level. However, depending upon the structure of the relevant libraries or applications, general-purpose dynamic update systems [2, 19, 24] may be more suitable or practical.

### 7.5 Implementation in other systems

As our approach has been to implement dynamic update within an existing OS with the aim of developing a design suitable for commodity operating systems, one goal for future work would be to apply it to a commodity OS. In previous work [5], we described how dynamic update may be implemented in Linux. Although it does not provide the same consistent mechanisms, modular parts of Linux such as the VFS layer and device driver interfaces are effectively object-oriented, providing data hiding and indirection. For example, filesystem drivers are invoked through a table of function pointers held in the *inode* structures, and device drivers are called similarly. This provides the same indirection as K42's object translation table, although at a coarser granularity. It would result in a lower overhead for indirection and state tracking, but potentially higher update costs. The design's other requirements, such as state tracking, can also be achieved within Linux, albeit not so consistently as in K42 [5].

One significant difference between K42 and Linux is blocking threads. K42 kernel threads are short-lived and non-blocking, allowing us to block access to an object and simply wait until existing threads terminate to achieve quiescence. In Linux, however, system call handlers may block for IO or other long-running operations, that we cannot wait for. Our current solution when applying an update in Linux is, when possible, to abort system calls with *EINTR* (interrupted call) or *EAGAIN* (resource temporarily unavailable) errors, from which an application can recover by retrying the call. If a blocking call cannot be interrupted, we must delay and retry the update until it completes. This could be avoided by a wrapper that converts blocking system calls into restartable variants such as *select*.

The main limitation in applying this approach to Linux is the current extent of modularity. Unlike K42, core parts of Linux such as the scheduler and virtual memory system are not modular. Despite this, we believe that dynamic update for Linux is feasible. In particular, it is not necessary to apply modularisation and dynamic update infrastructure throughout the kernel. Rather, dynamic update can be enabled incrementally for specific subsystems, by adding indirection and state tracking (or subverting existing structures). Along with other projects [26, 28], this provides motivation for increasing the modularity of the Linux kernel.

## 8 Related work

Many dynamic update systems exist for high-level languages [3, 10, 14, 20, 25], however these are inapplicable to an OS implemented in C or C++. A small number of general-purpose dynamic update systems for C have been described in the literature [2, 12, 19]. These focus on application code, however an OS kernel is a fundamentally different environment, and features constraints such as a high level of concurrency, and completely event-driven execution and control flow. Also, operating systems offer extremely limited runtime environments. These constraints result in different trade-offs and a different design for dynamic update.

Most general-purpose dynamic update systems for C do not support threading [12, 19]. One that does is OPUS [2], that uses Linux's *ptrace* facility to update C programs at function boundaries, with the goal of enabling dynamic security patches. OPUS waits for updated functions to be off the stack of all threads. Unlike our system, which blocks new invocations to achieve quiescence, it is possible for OPUS never to achieve quiescence, and thus for an update to be delayed indefinitely. Because OPUS relies on stopping all threads to examine their stacks, it would be difficult to apply the design to an operating system kernel, where thousands of threads may be present, and where blocking the whole system's execution is not feasible. OPUS does not handle changes to data structures nor function interfaces. Despite these limitations, it was able to apply many real security patches, suggesting that our system is also suitable for security patches.

LUCOS [8] is a dynamic update system for Linux built

upon the Xen virtual-machine monitor. To apply updates, LUCOS enforces quiescence by using the VMM to stop the system's execution; it then dynamically patches functions. In contrast, in K42 we block and quiesce only objects affected by an update, allowing the rest of the system to continue. Similarly to OPUS, LUCOS does not support changes to function interfaces, that are important even for bug fixes, as we have shown. If a LUCOS patch changes data structures, pages containing old and new versions of the affected data are marked read-only; on a write fault the kernel is single-stepped, and a data-transfer function is used to keep the versions consistent. K42 avoids such two-way data conversions by updating all code that accesses a data structure along with the data itself. For LUCOS to detect when functions using old data structures have returned, it must examine every stack frame of every kernel thread in the system, a large scalability problem on modern systems that commonly run thousands of threads. Because it only consists of passive modules, LUCOS' performance overhead is negligible at the expense of higher update costs, especially when data structures are changed. Our system incurs constant overhead from indirection and state tracking, although, as we have shown, this overhead is very low (less than 1%). Furthermore, our system's overhead compares favourably with virtualisation, and enables a simpler and more scalable update process.

DynAMOS [15] is another recent dynamic update system for Linux. Like LUCOS, it uses function-level binary patching, which occurs while interrupts are disabled. DynAMOS supports limited updates to data structures through the use of shadow structures, that must be maintained separately to the original structure. Like LUCOS, DynAMOS may need to walk the stack of every kernel thread to detect quiescence. As a loadable module that requires no modifications to the base kernel, DynAMOS does not impace base performance, however because every updated function incurs an extra indirect branch, the performance impact of updates is significant. Micro-benchmark results show overheads higher than 40% for some functions, but the overall performance impact of an update is not reported.

The main advantages of LUCOS and DynAMOS over our approach are that they do not require changes to the kernel, are able to update almost any function, and do not incur any base performance impact. However, due to the use of binary function patching as the underlying mechanism, the impact of applying updates in these systems is significantly higher. Although our approach has not yet been applied to Linux, the comparison between it and these systems is primarily a trade-off between applicability, in terms of the parts of the kernel that can be updated, and performance and complexity, in terms of the cost of applying updates, the difficulty in developing them, and the complexity of changes that are supported.

A practical approach to general-purpose dynamic update is taken by Ginseng [19]. Ginseng compiles C programs specially, adding indirections for types and functions, to provide safe, fine-grained dynamic updates for arbitrary C code. This contrasts to our approach, that applies updates at the coarser level of objects or modules, and so suffers lower overheads from indirection and update support, but relies on the modular structure of the OS. Ginseng does not support threaded execution, although the authors are currently investigating ways to cope with the concurrency requirements of an OS [18].

Several domain-specific approaches to dynamic update appear in the literature, for example in object databases [6] and distributed systems [1]. These systems are not necessarily language-based, but are closely tied to their domain for other reasons. Our approach is similar in that we have tailored the design of our system to the structure and environment of the code we update, namely modular kernel code.

One relevant domain-specific dynamic update system is Upstart [1], which provides automatic software updates for distributed systems by interposing at the library level and rewriting remote procedure calls. Despite the different focus, a lot of parallels can be drawn between Upstart and K42. Upstart's transform functions are equivalent to our state-transfer functions, simulation objects play the same role as adaptors, and scheduling objects are a generalisation of lazy update.

AutoPod [22] is a kernel module for Linux that provides checkpoint, migration and restart of processes transparently to the applications and kernel. Combining AutoPod with an underlying VMM, as long as the kernel interface is unchanged it is possible to start a new virtual machine with an updated kernel, checkpoint the user processes in the old virtual machine, and migrate and then restart them on the updated kernel. This represents a radically different approach to the problem; it avoids significant changes to the OS, at the cost of runtime overhead from virtualisation and AutoPod.

## 9 Conclusions

We have implemented dynamic update for K42, and found that adding this feature to an existing operating system is feasible when that system has a sufficiently modular structure. K42 has the advantage that it is object oriented with a consistent and pervasive module invocation mechanism, but as we have discussed, commodity operating systems such as Linux can provide the same support, albeit in an less consistent fashion.

We have shown how to design a dynamic update system that handles interface changes, which are required for many bug fixes, and applies updates lazily, which is

essential for changes to some data structures because it mitigates their severe performance impact.

From a study of the K42 revision history, we have shown that our dynamic update model can support at least 79% of maintenance changes to an operating system, and we expect that for a real system the proportion would be closer to 100%. We have also measured the performance impact, and found it to be insignificant, with less than 1% runtime overhead.

Dynamic update is a rich area for future systems research, and will become increasingly important for mainstream operating systems. We have shown one way to achieve a dynamic update feature in operating systems. With dynamic update, the uptime of systems should be limited by hardware failure, not by software updates.

## Acknowledgements

The K42 community provided valuable assistance—Raymond Fingas developed the *arbiter objects* used for lazy update and adaptors, and Jeremy Kerr implemented the *sync* changes. Comments and critical feedback were provided by the participants of the First EuroSys Authoring Workshop, Eno Thereska, Iulian Neamtiu, and the anonymous reviewers.

This work was partially supported by DARPA under contract NBCH30390004. National ICT Australia is funded by the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

## Notes

[1]We examined the *kitch-core* and *kitch-linux* CVS modules.

[2]In K42 terminology, we are referring to the file cache manager (FCM) and file representative (FR) objects.

## References

[1] S. Ajmani. *Automatic Software Upgrades for Distributed Systems*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Sep 2004. Also as Technical Report MIT-LCS-TR-1012.

[2] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online patches and updates for security. In *14th USENIX Security Symp.*, pages 287–302, Aug 2005.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*, chapter 9, pages 121–123. Prentice Hall, 2nd edition, 1996.

[4] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *2005 Ann. USENIX*, pages 279–291, Apr 2005.

[5] A. Baumann, J. Kerr, J. Appavoo, D. Da Silva, O. Krieger, and R. W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *6th Linux.Conf.Au*, Apr 2005.

[6] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, pages 403–417, Oct 2003.

[7] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *2004 Ann. USENIX*, pages 15–28, Jun 2004.

[8] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live updating operating systems using virtualization. In *2nd VEE*, pages 35–44, Jun 2006.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[10] S. Gilmore, D. Kírlí, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, Department of Computer Science, The University of Edinburgh, Dec 1997.

[11] P. Gray. Experts question Windows patch policy. ZDNet News, Nov 2003. http://news.zdnet.com/2100-1009_22-5105454.html.

[12] D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Softw.: Pract. & Exp.*, 23(9):949–964, Sep 1993.

[13] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *EuroSys Conf.*, pages 133–145, Apr 2006.

[14] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin-Madison, May 1983.

[15] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys Conf.*, Mar 2007.

[16] K. B. Mierle, K. Laven, S. T. Roweis, and G. V. Wilson. CVS data extraction and analysis: A case study. Technical Report UTML TR 2004-002, Dept of Computer Science, University of Toronto, Sep 2004.

[17] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *2nd MSR*, pages 2–6, May 2005.

[18] I. Neamtiu and M. Hicks. Dynamic software updating for the Linux kernel. OSDI, Work-in-Progress Session, Nov 2006. Seattle, WA, USA.

[19] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI*, Jun 2006.

[20] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Int. Conf. Softw. Maintenance*, Oct 2002.

[21] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In *EuroSys Conf.*, pages 59–71, Apr 2006.

[22] S. Potter and J. Nieh. Reducing downtime due to system maintenance and upgrades. In *19th LISA*, pages 47–62, Dec 2005.

[23] E. Rescorla. Security holes... who cares? In *12th USENIX Security Symp.*, pages 75–90, Aug 2003.

[24] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *Softw.*, 10(2):53–65, Mar 1993.

[25] D. Stewart and M. M. T. Chakravarty. Dynamic applications from the ground up. In *ACM SIGPLAN Haskell WS*, Sep 2005.

[26] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *6th OSDI*, Dec 2004.

[27] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *3rd OSDI*, pages 117–130, Feb 1999.

[28] E. Witchel, J. Rhee, and K. Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *20th SOSP*, Oct 2005.

[29] T. Woerner. Understanding the Linux kernel initcall mechanism, Oct 2006. http://geek.vtnet.ca/doc/initcall/.

[30] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *1st MSR*, May 2004.