Dynamic Update for Operating Systems

Doctor of Philosophy
School of Computer Science and Engineering
The University of New South Wales

Andrew Baumann

August 2007

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. The aim of this research is to develop a model for dynamic update of operating systems, allowing a system to be patched without the need for a reboot or other service interruption.

In this work, a model for dynamic update based on operating system modularity is developed and evaluated using a prototype implementation for the K42 operating system. The prototype is able to update kernel code and data structures, even when the interfaces between kernel modules change. When applying an update, at no point is the system's entire execution blocked, and there is no additional overhead after an update has been applied. The base runtime overhead is also very low. An analysis of the K42 revision history shows that approximately 79% of past performance and bug-fix changes to K42 could be converted to dynamic updates, and the proportion would be even higher if the changes were being developed for dynamic update. The model also extends to other systems such as Linux and BSD, that although structured modularly, are not strictly object-oriented like K42.

The experience with this approach shows that dynamic update for operating systems is feasible given a sufficiently-modular system structure, allows maintenance patches and updates to be applied without disruption, and need not constrain system performance.

Contents

List of Figures

List of Tables

Originality Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

. . . . . .
Andrew Baumann, 14th August 2007

Portions of this work were previously published in the following papers and articles:

- Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Improving operating system availability with dynamic update. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, USA, October 2004.
- Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 Annual USENIX Technical Conference*, pages 279–291, Anaheim, CA, USA, April 2005.
- Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, Australia, April 2005.
- Andrew Baumann and Jonathan Appavoo. Improving dynamic update for operating systems. In *Proceedings of the 20th ACM Symposium on OS Principles, Work-in-Progress Session*, Brighton, UK, October 2005.
- Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, Amos Waterland, David Tam, and Andrew Baumann. K42: an infrastructure for operating system research. *Operating Systems Review*, 40(2):34–42, April 2006.
- Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proceedings of the 2007 Annual USENIX Technical Conference*, pages 337–350, Santa Clara, CA, USA, June 2007.

1 Introduction

1.1 Problem statement

In many computing systems, downtime is becoming increasingly expensive. Users and clients expect high availability, and system administrators are tasked to provide it. Traditionally, system availability was limited by hardware failures. However, modern hardware offers redundancy at many levels, and software techniques such as virtualisation, with the ability to migrate virtual machines when hardware fails, mean that computing systems are limited not by hardware failures but by software. In particular, the operating system (OS) must also be highly available, because any failure or downtime affecting the OS prevents the entire system from functioning.

There are significant limitations to OS availability. A bug in the OS can critically affect the security, correctness or performance of the system. However, modern operating systems are large and complex pieces of software, and bugs are inevitably discovered after the OS is released. For critical bugs, the OS vendor will usually develop and release an update, or *patch*, for the problem. Most of these patches require a system restart to take effect.

Restarting a system, or rebooting it, results in downtime. Even if it is scheduled, such downtime can be so expensive that administrators are forced to trade its cost against the risks of remaining unpatched. Many administrators do not apply well-announced and widely-propagated security-critical patches for weeks [Res03]. Rebooting a system also causes loss of transient state, and thus may be a serious inconvenience to its users.

Further exacerbating the problem, major OS vendors are releasing an increasing volume of patches and are doing so more frequently than before [Gra03, FGC+05].

1.2 Aim and scope of research

Changing or patching a program on-the-fly, without affecting any of the services it provides, is known as *dynamic update* [SF93]. The aim of this research is to develop a model for *dynamic update of operating systems*.

One approach to achieving the aim might be to reimplement the operating system in a high-level dynamic language, or to start with a specialised OS written in such a language, because, as will be shown in Section 2.1, much of the existing work in dynamic update is based on dynamic languages. However, the motivating problems exist primarily in mainstream *commodity* operating systems, and although implementing a new OS that supports dynamic update might solve the problem, it would only have created the much larger problem of migrating all the existing applications to that system. For this reason, this research will focus on dynamic update mechanisms that could be applied within the context of commodity operating systems, for example, Windows or UNIX derivatives (such as Linux and BSD). These systems are built from large and complex bodies of code not amenable to major changes or restructuring. The goal of this work is therefore to provide dynamic update support within the operating system, using its existing features where possible, and maintaining its existing structure. However, unlike some other work that will be discussed in Section 2.3, the possibility of modifying the underlying OS where feasible to enable dynamic updates is not excluded.

Another important consideration is what kinds of updates must be supported. The problem that has been described concerns patches and updates created for maintenance purposes after an operating system has been released and is in use. It is desirable that any patch released for maintenance be dynamically updatable. This includes patches and bug fixes for correctness, performance and security. Occasionally maintenance releases might also add new features or functionality, however to take advantage of new features would also require updating and changing applications, and furthermore the time constraints for applying bug fix patches are much tighter than for new features, so they are not the main focus of this work. For many software projects, one could say that we wish to be able to update between minor revisions, but not necessarily across major releases. Using the Linux kernel as an example, an administrator might update a system from version 2.6.18 to version 2.6.18.1 and then 2.6.18.2.

One consequence of this choice is that a system will still eventually need to be rebooted. It is not clear, and it is not the aim of this research to determine, whether a commodity operating system could be built to be continually updatable. However, if a system must only be rebooted for hardware failure or major upgrades adding new features, the problem that currently forces administrators to choose between applying critical bug fixes or suffering expensive downtime will have been solved.

Finally, this research will consider only patches and updates to kernel code. Although this does not encompass all the code that might be considered part of an operating system (for example, user-level utilities and library code), kernel code is all security critical, and in commodity systems forms the bulk of the operating system. Furthermore, as will be discussed in Section 2.1, previous research has developed dynamic update systems for user-level code, and this could be adapted for use outside the kernel. Moreover, restarting user-level services is not as severe as rebooting the whole system.

1.3 Overview of this dissertation

Chapter 2 examines the literature on dynamic update systems and operating-system customisation. It points out the limitations of the existing work in enabling operating system dynamic update, that have mostly resulted from different aims. Chapter 3 notes that a combination of ideas from these two fields of research opens up the possibility of dynamically updatable operating systems. Based on this observation, a set of requirements for building a dynamically updatable operating system are developed, and a research method to test these requirements is presented.

Chapter 4 details a prototype implementation of dynamic update within a research operating system, called K42. The following two chapters report on the results obtained using this prototype. Chapter 5 presents results obtained from an analysis of the K42 source revision history, that is used to analyse the limitations and applicability of the prototype. A series of experiments to measure the performance characteristics and overhead of the dynamic update implementation is reported in Chapter 6.

Next, Chapter 7 discusses the issues in applying the design of the dynamic update prototype to other modular operating systems such as Linux and FreeBSD, and argues that it could be implemented in these systems. Finally, the results are are discussed and the model evaluated in Chapter 8, and conclusions are drawn and future work described in Chapter 9.

The primary contributions of this work are:

- a design for dynamic update in a modular operating system, including support for updates that change kernel code, data structures or module interfaces;
- a prototype implementation of the design using the hot-swapping facility in K42, that has negligible performance impact on the base system, and scales to work under high system loads;
- the use of laziness to enable dynamic updates of kernel data structures with many instances;
- an analysis of an operating system's revision history showing the type and frequency of maintenance changes, and the use of that analysis to determine the applicability of the dynamic update design;
- a discussion of how the developed model may be applied to commodity operating systems such as Linux and FreeBSD, that although modularly structured are not strictly object-oriented like the K42 system used in the case study, and do not already provide features such as hot-swapping.

2 Literature Review

In this chapter, the literature on two aspects related to dynamic update of operating systems is reviewed and discussed. First, dynamic update systems, a large number of which have been developed for application-level programs and some other domains such as databases and distributed systems, are covered. Next, research into making operating systems more dynamic for customisability, extensibility or adaptability is examined for its applicability to providing dynamic update features. Finally, direct research into the problem of dynamic updates to operating systems is discussed and critiqued in detail.

2.1 Dynamic update systems

The term *dynamic update* refers to performing software updates, to fix bugs or add features, without the need for downtime [SF93]. Dynamic update is also referred to as *on-the-fly program modification* [Fab76, HW96] or *on-line version change* [GJ93, GJB96]. An introduction to and overview of the field was given by Segal and Frieder [SF93], who identified general desired characteristics of update systems. The characteristics most relevant to this work are that update systems should:

- *preserve program correctness*, by performing updates only at correct times;
- *minimise human intervention*, or make the process of applying an update as automatic as possible;
- *support low-level program changes*, including changes to interfaces and data structures;
- *support code restructuring*, by allowing updates that alter the structure of the system rather than merely changes within previously-defined module boundaries;
- *not constrain the language and environment*, allowing existing programs to benefit from the dynamic update system.

A large number of dynamic update systems have been described in the literature. The majority of these systems are general-purpose and focus on adding dynamic update support to application code. As will be seen, they generally focus on achieving the first four

characteristics at the expense of constraining the language and system environment in ways that make them inapplicable to operating systems code.

Although it will not be considered further, one approach to the problem of dynamic updates is to use hardware support. A number of hardware-based systems have been developed for use in such devices as telecommunications switches. These systems usually contain duplicated central computers, each of which is capable of handling the normal workload. When a dynamic update is to take place one computer is upgraded first, and then the control logic directs requests to the new computer [SF93]. The specialist hardware required makes these systems costly and not generally applicable. Furthermore, in situations where the application maintains state information that cannot be discarded at update time, hardware support alone is insufficient.

Common concerns

Any software-based dynamic update system faces a number of problems and design choices that are fundamental to the domain. These are outlined here, and will be used to discuss and categorise the update systems examined later in this chapter.

First, any update system has a *unit of change*. This is the smallest structural unit of the program which is changed by an update, and determines the granularity of updates. In many systems, the unit of change is a function or procedure, but other units of change are server processes, modules, objects, abstract data types, or even individual language definitions. The choice of the unit of change is largely dictated by the structure of the system to be updated; for example, in update systems for procedural languages, function-level update is common, whereas most dynamic update systems for Java apply updates at the class level. The selection of a unit of change significantly impacts the other concerns that follow. It is also related to whether a unit's *interfaces can be changed*, for example function prototypes or method signatures.

An important limitation of a dynamic update system is *when updates can be applied*. This is usually determined by the unit of change and the need to preserve program correctness. For example, in systems that update procedural code at the function level, it is common to restrict updates to be applied only when none of the affected functions are being executed. Closely related to the question of when updates can be applied is *what code can be changed* by an update. In the previous example of updates only to inactive functions, long-running or top-level functions may never be updated. Also, and of particular concern to operating systems, low-level code such as event handlers or assembly stubs may now be handled by a dynamic update system.

Updating data, especially non-local variables, complicates the selection of safe times to apply updates, and makes maintaining program correctness significantly more difficult. Hence, some dynamic update systems do not support any changes to data structures,

while others allow changes only to certain kinds of data such as local variables, or only allow data to change in certain ways (for example, the size of a data structure may be fixed). Thus, another important property of an update system is *how data structures can be updated*. If updates to data are limited, the practicality and usability of the dynamic update system will be limited to the frequency of updates that must change data. If updates to data are supported, then when an update is applied, the system must convert existing data to the new format; this is usually performed by programmer-supplied *state or type transformer functions* that are run at update time.

Finally, the level of *runtime support* required by the dynamic update system is important, especially for operating systems. Some update systems rely on dynamic memory allocation and garbage collection, others require dynamic type checking. In restricted execution environments, these systems are unusable. This issue will be expanded on in the following subsection.

Limitations on operating-systems code

Operating-system source code, and in particular kernel code, has unique constraints not found in application-level source code that limit the applicability of general-purpose dynamic update systems to it. Commodity operating system kernels are implemented in a combination of C (or occasionally a subset of C++) and assembly language, and execute in a restricted runtime environment; specifically:

- Because the kernel primarily services requests from applications and hardware devices, it is mostly event-driven and has a high level of concurrency not usually found in application programs. Hardware interrupts must be serviced in a timely fashion, and their arrival may preempt other kernel code.
- Programmers must take particular care with locking. In some code (for example, interrupt handlers), no blocking operations may be performed.
- Dynamic allocation of memory is usually limited, and is not possible at all in some circumstances.
- Kernel code executes on a stack of a limited size. Therefore, deep nesting of functions and programming techniques such as recursion may lead to a crash.

These constraints will be used to discuss the requirements and limitations of the dynamic update systems examined in this chapter.

What must be updated

Neamtiu *et al.* conducted a study of source code evolution in several common open-source software packages with the goal of informing the design of dynamic update systems [NFH05]. From an analysis of the evolution of the Vsftpd FTP daemon, OpenSSH server, Apache web server and Linux kernel, 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 [PLM06] highlighted the problems associated with changes to interfaces in that system. Finally, the importance of changes to interfaces and data structures is also confirmed by a new analysis of a research operating system's revision history, the results of which will be presented in Chapter 5. Therefore, the dynamic update systems examined in this review will be assessed for their ability to support such changes.

2.1.1 Formalising dynamic update

Some attempts have been made to formalise the properties and semantics of dynamic update; they are described here.

For a simple model of procedural program execution, Gupta *et al.* proved by reduction to the halting problem that the question of whether it is safe to apply an update at a given point in a program's execution is in general undecidable [GJB96, Gup94]. That is, it is impossible to find all the points in program execution at which a program may safely be updated, but using conservative choices it is generally possible to find a subset. Gupta's dynamic update system, which is based on state transfer between processes, will be described in Section 2.1.4.

Duggan has developed a type-based approach and calculus for updating modules in a running program [Dug01]. Duggan's approach depends on dynamic typing, and converts data lazily between old and new types in either direction during program execution. Related to Duggan's work is Dynamic ML [GKW97, WKG00], a proposed implementation of a similar mechanism in a variant of the ML language, however in Dynamic ML a modified garbage collector is used to perform all type conversions simultaneously rather than lazily on access.

Bierman *et al.* developed an abstract calculus for dynamic update with the goal of enabling formal reasoning about how and when to ensure update safety [BHS+03]. Their update calculus is applied to a simple model of a server program and used to reason about its safety. Proteus [SHB+05] is a formal calculus for dynamic update in C-like languages by the same authors. For increased flexibility, it supports changes to function and data definitions, and to named types (such as C's *structs* or *unions*). To achieve safety, Proteus

uses a static updateability analysis that can determine at compile time what types can be safely updated at pre-defined update points in the program. These update points could be inserted by the programmer or automatically by the compiler. An implementation of the updateability analysis is used as the basis of Ginseng, a dynamic update system for C programs that will be described in Section 2.1.4.

2.1.2 Early software systems

Multics

Multics [Org72] was an early and highly-influential time-sharing system, that was developed starting in 1965 and aimed to provide computing services as a utility, akin to electricity or the telephone system. To meet this goal, it needed to be highly available, and to support online updates. The hardware used was hence reconfigurable [Sch71], as was much of the system software.

A form of dynamic linking was used to enable dynamic updates in Multics [Hon81, section 3]. Every function call was made indirectly, via a pointer in the *linkage segment* [DD68]. When a function was updated, the relevant linkage segment entries could be marked as invalid, forcing a trap and transparent relink to the new version of the function on its next invocation.

This mechanism was used to enable function-level updates of system and user software. As of 1972, Corbató *et al.* reported:

> "...it has been possible to maintain steadily for the last year or so a pace of installing 5 or 10 new or modified system modules a day. Some three-quarters of these changes can be installed while the system is in operation. The remainder, pertaining to the central supervisor, are installed in batches once or twice a week." [CSC72]

However, as this quotation shows, Multics lacked a mechanism for updating code in its central supervisor (which would today be termed the kernel). It also did not provide any specific support for changes to function interfaces or data structures.

Fabry's dynamic type replacement

Fabry [Fab76] provides another early description of a software dynamic update system. Motivated by the needs of large database systems, Fabry's scheme adds a level of indirection on module invocations which allows the calls to be redirected and an update

performed. It also lazily updates data structures as they are referenced through the use of a version field and multiple-reader/single-writer lock on each structure. However, it requires a capability-based addressing scheme, which limits its applicability.

DAS

The dynamically alterable system (DAS) was a research operating system developed at the Technical University of Berlin [GIL78]. Like Fabry's system, DAS supports changing the implementation of a module with a fixed interface. Although it does not require capability-based addressing, it does rely on a special mechanism provided by the DAS kernel—modules each reside in their own address space segment, and are invoked through a cross-address-space invocation mechanism. This mechanism was implemented in the microcode of a PDP 11/40E, and would be ill-suited to current CPU architectures. It was also not possible to update the kernel.

The designers of DAS recognised the problem of converting state information from the format used by an old module to the format used by the new module which is to replace it. Whereas Fabry required the new module to include conversion routines from the old format, DAS supports arbitrary upgrades by requiring that all modules implement a restructure operation, whereby the old module reconstructs its state in the new module using the common module API. This restructuring occurs at upgrade time.

DYMOS

Both Fabry's scheme and DAS suffer from some severe limitations. The interface of each module is fixed and cannot be changed by an upgrade, nor can modules be added or removed. These limitations are addressed by the DYMOS system [Lee83], which was designed to automate and simplify the process of creating and applying software patches, and dynamically updates procedures within a running program. Since DYMOS supports changing the interface of a procedure, it must also support changing multiple procedures in a single operation (to allow for changes to the places where a procedure is invoked).

The system allows the programmer to simply modify the source of the program, as they would normally do in fixing a bug or developing a new feature. Tools are then used which semi-automatically determine the changes from the running system, and produce the appropriate patch data including conversion functions. The tools also perform consistency checks, for example checking that if the specification of a module has been changed, all places where that module is invoked have been updated. When performing the update, the programmer is required to specify (in terms of procedures that must be idle) the conditions required to perform the update. The updating system waits for these conditions to be met before the update occurs.

The main drawback of DYMOS is that it requires programs to be written in an obscure language (StarMod [Coo80], a distributed variant of Modula).

2.1.3 High-level languages

In dynamically-typed languages, data values carry type information that is checked by the runtime system when data is used or functions invoked. This makes the task of a dynamic update system easier, because it is possible to change the type of data and code at runtime without fundamental changes to the system. The classic example of dynamic language support for updates, *Erlang*, is a concurrent functional programming language designed for large-scale industrial systems with real-time requirements, that was initially developed by Ericsson for use in telephone switches. It includes support for dynamic code updating, whereby it is possible to compile and load a new version of a module into a running system [AVW+96]. After an update, the next time the module is invoked, the new version will be used. Because Erlang is dynamically typed, the problem of interface changes and state conversion can be left to the programmer, although if the programmer neglects to update a data item, or calls a function with the wrong arguments, an exception will be generated. Similarly, the *Smalltalk* language used in the Resilient System [ABG+04] enables dynamic rebinding of method calls, and has been used to implement dynamic update in embedded systems applications.

Dynamic ML [GKW97, WKG00] is a modified version of Standard ML (a statically-typed compiled functional language) that supports run-time replacement of modules. A replaced module must have exactly the same interface and type signature as its predecessor, although data encapsulated within the module may be changed, making the system ideal for updating programs that make use of abstract data types. Because of this strict limitation, the authors avoided adding dynamic type information to the Standard ML run-time system. Module replacement is performed by a modified garbage collector; each instance of the affected type is converted in a single pass, avoiding any problems caused by different versions of a module co-existing.

A number of dynamic update systems have been developed for *Java*. Sun's HotSpot JVM includes limited support for hot-swapping classes [Sun02, pages 15–16]; using the debugging interface, a user may replace the implementation of methods in a class, however there is no support for changes to class interfaces or data.

The Java distributed run-time update management system (JDRUMS) is a modified JVM with reflection and dynamic update features [RA00,DH01]. To perform a dynamic update in JDRUMS, a developer implements a special Java class that has subclasses mirroring the old and new implementations of the class to be updated, and is responsible for performing state transfer between objects. Once an update is loaded into the JVM (which can occur across a network), old object instances are detected and converted lazily on access.

Gustavsson identified a number of shortcomings of the dynamic update support in JDRUMS [Gus01]. Because JDRUMS requires a strict one-to-one mapping between old and new classes and objects, and as the conversion functions have no access to methods or any other objects beside that being updated, the scope for restructuring the class hierarchy is very limited. Furthermore, classes active on the stack of any thread cannot be updated, and the supporting tools for building dynamic updates are limited. Gustavsson proposes to address some of these problems by switching from the use of Java to a dedicated and more flexible specification language for dynamic updates.

In the lazy functional programming language *Haskell*, Stewart and Chakravarty demonstrated how dynamic linking can be used to build an application so that it is reconfigurable and dynamically updatable [SC05], however the programmer must structure the application in a particular manner, and implement support for loading updates and coordinating the reconfiguration.

The biggest problem with all language-based approaches is that they require updatable programs to be implemented in a specific language. This means that programmers must be retrained, but also that existing programs cannot be made updatable without rewriting them. Most importantly for the aims of this work, it rules out their application to a general-purpose operating system implemented in a lower-level language.

2.1.4 Dynamic update for C and C++

The problem with the previous approaches is obviously that they are limited to applications implemented in the respective languages. However, because this work is concerned with support for dynamic upgrade of operating system components, that for performance reasons must be able to be implemented in low-level languages such as C, a language-based approach is inappropriate. A number of general-purpose dynamic update systems for applications written in C or C-like languages have been developed; they are described in this section. The descriptions of the different systems will be followed by a comparison and discussion of their applicability to operating systems code.

Gupta

Gupta's update system [GJ93, Gup94] works on C programs at function boundaries. For a function to be updated, there must be no references to it on the stack; this prevents the application of updates to long-running functions such as main(). Furthermore, changes to data are limited to local variables.

Gupta's system uses the novel approach of state transfer between processes. The implementation works by transferring the state of the old program into a new process with the

updated program's code segment. State transfer involves copying the register state, stack and data segments from the old process, and modifying the program counter and any return addresses on the stack to match the new code segment.

To transfer implicit kernel-level state, programs must use wrapper functions for system calls, allowing this state to be recorded and recreated in the new process. For example, these wrappers reopen any files in use by the old process, and seek to the same file offset. This mechanism is unique, but it has a number of unresolved problems. Most significantly, not all kernel-level state can be transferred as simply as file descriptors. Network sockets, for example, cannot be closed and reopened transparently. Furthermore, the process ID of the program will change, which may affect other processes interacting with it.

OPUS

On-line patches and updates for security (OPUS) [ABB+05] is a dynamic update system for C programs specifically targeting security patches. In regards to the types of update that it supports, OPUS is quite similar to Gupta's system. It updates C programs at function boundaries by taking control of the running process, waiting until none of the functions to be updated are active on any thread's stack, and using dynamic linking and binary patching techniques to insert new code for the changed functions. As a result, its limitations are very similar: global data structures may not be changed, and long-running and top-level functions cannot be updated. Furthermore, function interfaces are also fixed.

Where OPUS improves over Gupta's work is in the process of creating dynamic patches. OPUS uses a modified version of the GCC compiler, and can therefore easily be integrated into an application's normal build process. A dynamic patch is created simply by compiling the original system and the modified system, and running a tool over the two build trees. Static analysis is used to detect changes that may be unsafe (for example, changes that modify non-local program state) and alert the programmer. Although the system is quite limited in what it can update, the authors believe that this is sufficient for most security patches—they were able to apply updates for 22 of 26 hand-tested CERT vulnerabilities.

Dynamic C++ classes

Hjálmtýsson and Gray have developed a system supporting dynamic replacement of C++ class implementations in a running program [HG98]. This system works with standard C++ compilers through the use of automatically-generated proxy classes, which forward method invocations to the currently active implementation.

The dynamic classes system has the significant advantage that it can be added to existing programs written in C++. However, the system does not support changes to class interfaces (since the proxy classes are fixed at the initial compile time). It also avoids the problem of transferring state information by choosing to keep old classes in existence after an update—no data is converted, and only new instances of updated classes run the updated code.

Although it works with C++, this approach is quite different to the other dynamic update systems discussed in this section. Rather than attempting to support changes anywhere in a program, it only allows changes that are within a dynamic class and do not change the existing class interface. Furthermore, because existing class instances are not updated, long-lived objects will not be affected by any dynamic changes. In this regard, the system is closer to a dynamic linker for C++ than a general-purpose dynamic update mechanism. This limits its use for applications such as security patches, where it may be critical that existing objects are also updated.

Hicks

Hicks [Hic01, HN05] attempts to address many of the limitations of previous dynamic update systems, namely the flexibility of when and how programs may be changed, the robustness and correctness of the updated code, the ease of use of the system, and the overheads imposed by the system. The two main features of Hicks' system are the use of verifiable native code to ensure correctness, and the automatic construction of patches, using tools somewhat similar to the DYMOS system discussed previously in Section 2.1.2.

Patches in this system consist of typed assembly language (TAL), which means that a patch can be verified to not crash the system, and to have important properties such as type safety. Unfortunately, in order to be compiled to TAL, the program must be written in a type-safe variant of C known as Popcorn. This special language requirement is the system's greatest limitation.

Besides the use of TAL, the system is implemented primarily as a modified dynamic loader and linker, along with a tool to aid in automatic patch construction. The system achieves low overhead, 0.3–0.9% in throughput benchmarks of an updatable web server, although the raw overhead will depend on the workload and type of application.

Ginseng

Ginseng [NHS+06] contains an implementation of the static updateability analysis from Proteus [SHB+05] (discussed previously in Section 2.1.1), that is used to determine and insert safe update points into a C program. Pending updates are applied when the running

System	Language	Granularity	Supports threads	Function signatures	Local data	Global data	Code on stack	Data on stack
Gupta [GJ93, Gup94]	C	function	×	√	√	×[a]	×	×
OPUS [ABB+05]	C	function	√	×	√	×	×	×
Dynamic C++ Classes [HG98]	C++	object	√	×	×	×[b]	×	×
Hicks [Hic01, HN05]	Popcorn[c]	definition[d]	√	√	√	√	√	×
Ginseng [NHS+06]	C	definition[d]	×	√	√	√	√	√

[a] Global variables can be added, but only if provision has been made in the previous version.
[b] A new class may use different data structures, but existing object instances are not updated.
[c] Popcorn is a type-safe variant of C.
[d] These systems perform updates at the granularity of language definitions (functions, variables, types, etc.).

Table 2.1: Comparison of dynamic update systems for C/C++

program reaches a safe update point. At compile time, Ginseng also automatically adds indirections for types and functions to enable later updates to them.

Ginseng provides safe, fine-grained dynamic updates for arbitrary C code, with a performance overhead measured between 0 and 32%. It does not support threaded execution, and although the authors have discussed adding it in future work, this would require either extending the static analysis to handle all possible thread interleavings, or blocking threads at safe update points. Other work in progress includes an attempt at applying the tool to the Linux kernel, that will be discussed in Section 2.3.3.

Discussion

Table 2.1 summarises the features of different updating systems for C-like languages.

Both Gupta's system and OPUS are quite limited in the changes they can support; in particular, changes to data structures are very heavily restricted. Secondly, Gupta's model for process state transfer does not apply to a kernel. Furthermore, both require that a function be off the stack before it can be updated. In an operating system, this would require scanning the stack of every kernel thread to check if an update could be applied, potentially a huge performance and scalability problem.

Dynamic C++ Classes is a relatively simple approach that could work in a modularised operating system using C++, although it implies some constant runtime overhead from

the use of proxy classes. However, on its own this system does not provide true dynamic update features, because it does not convert existing objects, and does not solve the problems of achieving a safe point at which to do so.

Ginseng is the most general system, with its ability to support updates to C, including changes to code and data. However, it does not support threading, and adding such support is non-trivial. Furthermore, the cost of indirection on all kernel functions and data structures would be significant. Hicks' system supports threading while retaining the most important features of Ginseng, but it does not support C and instead requires programs to be compilable to typed assembly language. Converting an operating system kernel to a type-safe language, with its significant use of pointers, virtual memory operations and inline assembly code, would be a huge undertaking.

2.1.5 Domain-specific update systems

Client-server and distributed systems

In client-server and distributed systems, where communication is via message passing, performing an update can be achieved by redirecting the messages destined for one module (or server) to its replacement. Some of the work in this area includes that of Ajmani, whose Upstart system provides software updates for distributed systems by interposing at the library level and rewriting remote procedure calls [ALS03, Ajm04, ALS06], Bloom, who describes a dynamic update system for the Argus distributed system [Blo83], and Hauptmann and Wasel, who have developed a software reconfiguration mechanism based on the Chorus distributed operating system [HW96].

These approaches, while similar to other dynamic update systems, rely on the unique features of the underlying system. Bloom's work relies on the crash recovery and persistent data mechanisms of Argus, and Hauptmann and Wasel's approach relies on Chorus' ability to migrate message ports. Ajmani's work is slightly more general—it does not rely on a specific operating system, but instead interposes on remote procedure calls at the level of the standard sockets library, allowing its use with most distributed programs implemented using that library.

Persistent object systems

Persistent object stores, also known as object-oriented databases, are extensions of programming languages such as Java. They allow language data structures to persist beyond the lifetime of a program without the programmer explicitly serialising the data in permanent storage such as files or databases. Because the persistent storage format is closely

linked to the code and data structures of the application, there is a need to support upgrades and evolution of the persistent objects.

Boyapati *et al.* have developed an approach and prototype implementation for performing modular upgrades to persistent object stores [BLS+03]. Upgrades in their system consist of new versions of one or more classes, and for each class a *transform function* that converts the old version of an object's state to the format required by the new class. Upgrades are modular, which means that when a transform function runs it encounters only object interfaces that existed when it was written, allowing transform functions to be reasoned about in the same way as other methods. Because an upgrade may affect a huge number of objects on slow secondary storage, and to avoid stopping the execution of the entire system while an upgrade is applied, objects are upgraded lazily just before they are accessed by an application.

2.2 Operating-system customisation

In this section research specific to operating systems that might be used to enable dynamic update is discussed. Rather than the dynamic update systems covered in the previous section, where the primary problem being considered was how that work might be adapted to work inside an operating system, in this section the main question is whether the work could be used or adapted to enable dynamic update. Three main areas of operating systems research are discussed: the trend towards modularity in operating systems, research into customisable and extensible operating systems, and various approaches to dynamically patching a running kernel.

2.2.1 Modularity in operating systems

The advantages of using modularity to hide design decisions and encapsulate complex data structures in software systems are well established [Par72]. Modularity has long been used in the implementation of operating system kernels, with code including device drivers, the virtual memory system, the file system or the network stack being implemented as separate modules. Traditionally the set of modules in the kernel could only be changed at compile or link time, and required a reboot to take effect. However, the development of loadable kernel modules [Dra93] removed this requirement, allowing some classes of module (most commonly device drivers and file systems [Kle86]) to be loaded at run time. Most modern operating systems support loadable kernel modules [Sun91, SGI94, Rub98, dGdS99, Rei00].

As well as adaptation, software engineering concerns such as reliability and stability are another motivation for kernel modules. For example, the Nooks [SBL03, SAB+04] and

Mondriaan [WRA05] projects both improve the reliability of the Linux kernel by isolating modules and protecting the rest of kernel from some of the crashes that could be caused by bugs in their code. Finally, other researchers have observed that "fine-grain modularity within the kernel significantly increases the scope for performance optimisation" [PW93].

Loadable kernel modules enable a very limited form of dynamic update. A module may be unloaded and a new version loaded in its place. However, a module can only be unloaded when there are no outstanding references to it. For example, before unloading a device driver all access to the device must be stopped, and before unloading a file system module any file systems using it must be unmounted. This prevents updates from being applied without denying users access to some of the system's services. Another disadvantages of kernel modules for dynamic update is that they are limited to certain parts of the kernel. For example, in Linux 2.6 only file systems, device drivers, parts of the network stack, security modules and library functions such as cryptography are modularised, whereas key parts of the kernel including the virtual memory system and scheduler are not.

SOS is an operating system for sensor nodes that includes a module-based update scheme [HKS+05]. SOS consists of a small kernel and loadable modules that access kernel functions through a jump table, allowing the kernel implementation to be upgraded as long as the structure of the table and function interfaces is unchanged. Although is does not provide protected memory or other features expected of traditional operating systems, SOS shows how a simple dynamic update system can be built into a modular kernel structure.

Systems based on microkernels [HEV+98, HBB+98, GJP+00] have taken the concept of kernel modularity further, by moving as much code as possible out of the kernel and into separate user processes where it can be protected and isolated against failure. Although it is in some senses orthogonal to dynamic update concerns, the strict separation enforced by microkernel-based systems leads to clearer module boundaries and thus can simplify the implementation of dynamic update features. MINIX 3 [HBG+06] is a microkernel-based system that is also self-repairing; if a system component such as a driver crashes, it is restarted and the rest of the system recovers from the failure. This level of fault-tolerance would greatly simplify the implementation of dynamic update features, as an update can be considered a special case of failure. However, as the goal of this work is to develop dynamic-update techniques appropriate for commodity operating systems, microkernel-based systems will not be a further focus of this review.

2.2.2 Customisable and extensible operating systems

Customisability, extensibility and adaptability have been important areas of recent research in operating systems; a survey of research into such systems was provided by Denys *et al.* [DPM02]. A highly customisable operating system would support dynamic

changes between policies and features and allow new policies and services to be loaded into the running system, and therefore would provide many of the same features as a system supporting dynamic update. In this section, the contributions of customisable and extensible operating systems research to achieving dynamic update are summarised and discussed.

Modifications to existing systems

SLIC is an extensibility system for commodity operating systems that is based on the interposition and redirection of events crossing the user-kernel interface [GPR+98]. A prototype was implemented for Solaris on the SPARC architecture, that operates by rewriting the system call table and binary patching parts of the kernel such as the signal delivery code. Extensions may be loaded at user or kernel level, and can modify or block events such as system calls or signals. To evaluate the system, three extensions were developed: a fix for a CERT security advisory, an encrypted file system, and a restricted execution environment. These extensions embody the limitations of the system—because it targets unmodified commodity operating systems, SLIC can only alter interactions around the fixed system call interface, and thus is limited to extensions that work within that constraint. The CERT advisory extension fixes a security bug in a user-space application by monitoring the file operations of other applications. It is doubtful that security flaws in the kernel itself could be corrected by such a mechanism. Furthermore, the ability of the system to enable dynamic updates will be limited to modifying the behaviour of the kernel as perceived by user applications, preventing dynamic updates from being developed in the same way as normal kernel changes, and more importantly preventing dynamic updates that fix bugs in code that does not directly interact with user applications, such as the network stack or device drivers.

The *Synthetix* project attempted incremental specialisation [PAB+95], re-implementing parts of a commodity operating system to dynamically use specialised implementations for better performance. In a prototype implementation, the path of the *read* system call in HP-UX was modified to support a specialised implementation in the case where no other process has the same file open, avoiding the costly acquisition of concurrency locks and significantly improving the performance in the common case. To enable switching between the specialised and default implementations of the system call, a *replugging* facility was developed that enables low-overhead indirection for specialisable functions while maintaining safety properties in the presence of concurrent execution and replugging operations [CAK+96]. Although the goals of this work were adaptability rather than extensibility, the dynamic structures used are relevant to dynamic update, and the same replugging mechanism could be used to enable dynamic updates to kernel functions invoked through replugging points. However, it is not clear how much work was required

to implement the specialisation within HP-UX, nor how amenable other commodity operating systems are to the incremental modifications of this approach.

New system structures for extensibility

Many research projects have investigated using new or different operating system structures to achieve extensibility. Although many of these systems might be modified to support dynamic update, the result would not achieve the aim of dynamic updates to commodity operating systems. However, three systems with dynamic kernel architectures, Kea, Pebble and MMLite, have explicitly explored or discussed support for dynamic update, and are covered here.

Kea is a dynamic and extensible operating system based on a microkernel [VH96], however unlike the traditional message-passing IPC model offered by most microkernels, Kea uses *portals*. A portal is a special entry point that is invoked via a kernel-mediated transfer of the calling thread into the portal's protection domain. Unlike other portal-based systems, Kea allows portals to be remapped transparently to the user of a portal, enabling dynamic reconfiguration of services and potentially dynamic update. A service migration mechanism, which could also be used for service replacement, has been implemented for Kea based on portal remapping and state transfer; the authors claim that measurements of its performance show the "practicality of dynamic kernel upgrades" [VH98].

Another portal-based microkernel very similar to Kea is *Pebble* [GSB+99], although unlike Kea it dynamically generates specialised portal invocation code for performance reasons. Most system functionality in Pebble may be replaced dynamically by modifying portal tables, and Gabber *et al.* have discussed the applicability of Pebble to non-stop systems and its support for dynamic updates.

MMLite is a heavily componentised operating system architecture [HF98]. Components in MMLite are implemented as objects that can be dynamically loaded and assembled into a full system. Traditional system services such as virtual memory, name-space management, IPC, and device drivers are all implemented as objects. Objects can transparently be replaced while in use via a mechanism known as *mutation*. In a mutation, which allows changes to the implementation of an object, a mutator thread coordinates the execution of other threads accessing the object using appropriate synchronisation mechanism, and converts the state of the object to the representation expected by the new implementation. It is not clear whether changes to object interfaces are possible, or whether it is the responsibility of the object implementer to determine the *clean points* at which mutation may occur. The authors have briefly discussed the use of mutation to achieve dynamic updates, however do not appear to have explored this option any further.

K42 and hot swapping

K42 is an experimental object-oriented operating system supporting *hot swapping*, that allows the safe and dynamic replacement of object instances with different implementations [AHS+02, SAH+03, AHS+03]. K42 forms the basis for most of the experimental work in this thesis, and full details of the system will be given in Section 4.1, however for completeness an overview of the previous work on hot swapping is included here.

Every object in K42 uses virtual methods, and is accessed indirectly via a common object translation table, allowing the implementation and representation of a specific object instance to be changed by writing to the object's translation table entry. To perform this change safely in a concurrent environment, the implementation of hot swapping uses a thread-generation counter to detect quiescence in the relevant object before it is swapped, as will be described in detail in Section 4.1.4. During the quiescent period, the object's state information is transferred to the new instance, changing its representation if required by the new implementation of the class.

Hot swapping was used to achieve adaptability in K42, including: swapping between code optimised for access on a single node and code optimised for scalability, swapping between a general purpose implementation of a file handle and one optimised for the case where only a single process accesses that file, and swapping between least-recently and most-recently-used page replacement algorithms when sequential memory accesses are detected. Hot swapping could be used as the basis of a dynamic update mechanism, and Soules *et al.* discussed this possibility [SAH+03], but several important requirements are missing. First, there must be a way to load the code for an updated object implementation into the kernel, and second, once an update for a particular class is loaded a means of locating and updating every object instance of that class is required. Furthermore, hot swapping in K42 does not support changes to class interfaces, potentially limiting the updates that may be applied.

Moore describes a hierarchical-table-based mechanism for implementing indirection on all function calls within an operating system [Moo04]. This might be used in the implementation of a customisable or extensible operating system, and is an alternative structure to K42's flat object translation table, but does not in itself address many of the other problems in dynamic update, such as achieving quiescence before an update, or supporting changes to data structures or function interfaces.

2.2.3 Dynamic kernel patching

Dynamic kernel instrumentation tools are able to alter the flow of control in a running kernel for the purposes of debugging and instrumentation, usually by inserting jump or

trap instructions. However, they do not handle changes to data structures or function interfaces, nor consider dynamic update and code evolution. In this section the applicability of these mechanisms to providing dynamic update is discussed.

Function-level tracing and replacement

The dynamic kernel modifier (DKM) is a Linux kernel module that can insert trace points at the beginning of kernel functions, nullify functions, and replace functions with a different implementation [Min02]. It works by modifying function prologues (which have a predictable series of instructions) to jump out to stub code that performs the actions of the overwritten prologue instructions before logging a trace-point or performing some other activity. It is not clear how DKM avoids the inherent race condition in modifying multiple instructions to insert a jump to the stub code.

DKM's goal was to enable faster development and debugging of the kernel, but it might also be altered for use as a dynamic update mechanism. Because it can only replace existing functions, cannot change their interfaces, and cannot change any non-local variables or data structures, DKM's limitations as a dynamic update system would be very similar to those of the OPUS system discussed in Section 2.1.4.

DProbes is a similar mechanism for dynamic insertion of trace points in the Linux kernel [Moo01], however because it uses a single trap instruction rather than rewriting function prologues, it cannot replace function implementations and thus cannot be used for dynamic update. The same is true for the function boundary tracing facility in Solaris DTrace [CSL04], which replaces a single instruction in function prologues with a branch or trap, depending on the architecture.

KernInst: dynamic code splicing

KernInst provides dynamic code-splicing at almost any point in a running kernel [TM99]. A prototype was implemented for Solaris on the SPARC architecture as a combination of a loadable kernel module and user-space daemon. It uses an automatic analysis of the kernel machine code to determine a control flow graph and register liveness information. Code to be spliced is generated dynamically to use free registers if available or to automatically save and restore registers as required. For safety in multi-threaded kernels, only one instruction is replaced for any splice. Because the range of a single branch instruction is limited in RISC architectures, KernInst uses springboards—these are small regions of available memory within the branch-range of the original splice point that contain multiple-instruction branches to the splice code.

It is not clear how well the KernInst approach would work on architectures with variable-length instructions, such as x86. Because the jump instruction is five bytes, it may over-

write multiple smaller instructions at the branch point; in these situations it would be necessary to instead use a trap instruction and place recovery code in the fault handler.

Although the motivation of KernInst was to provide fine-grained dynamic instrumentation, the code-splicing technique might be altered for use as a simple dynamic update mechanism. The dynamic code generator would need to allow for registers to be modified in the original code, and for a splice to potentially return to a different point from which it was entered. However, the process of generating dynamic patches would be quite complex, the system would have limited scope because no data structures could safely be changed, and the performance of the system would degrade each time an update was loaded due to the overhead of branching to and from the spliced code.

2.3 Dynamic update to operating systems

A small number of researchers have directly attacked the problem of dynamic updates to operating systems. The resulting four systems, LUCOS, DynAMOS, Ginseng and AutoPod are discussed in detail here. Although working on the same problem, they represent very different approaches to solving it, and the strengths and weaknesses of the approaches are critiqued.

Solaris Live Upgrade [Sun01] is related to dynamic update in name only. This feature allows changes to be made and tested without affecting the running system, but requires a reboot for changes to take effect. It reduces the likelihood that a change has negative effects, but does not address the problem of reboots and system downtime. The same results could be achieved using virtualisation.

2.3.1 LUCOS

LUCOS is a dynamic update system for Linux built upon virtualisation [CCZ+06]. It consists of modules in the Xen virtual-machine monitor (VMM) and Linux kernel, and a user-level daemon. LUCOS applies binary patches at function-level, and is designed to avoid the need for achieving quiescence within the kernel. Instead, it patches active functions immediately by using the VMM to halt the entire system's execution, effectively enforcing quiescence by blocking events from a lower level. After the system is resumed from a patch, new calls to updated functions will use the new code, but old versions active on the stack will continue to execute until they return. Because it performs function-level binary patching, LUCOS does not support changes to function signatures.

LUCOS supports changes to data structures even when there are active references to old versions of the data structure present in old functions. This is achieved by maintaining both old and new versions of an affected data structure in read-only pages; on a write fault

to one of these pages, the kernel's execution is single-stepped, and every time a write to either the old or new version of a data structure is detected, a data-transfer function is invoked to keep the two versions consistent. LUCOS marks the pages read-write and resumes normal execution after all functions accessing the old data structure have returned; to detect this, it must examine every stack frame of every kernel thread in the system.

The combination of single-stepping access to pages containing updated data structures and scanning every kernel thread's stack leads to serious scalability problems and performance overheads during update application, especially if updating a data structure with many instances. In the reported experimental results [CCZ+06], only two of the five updates evaluated involved changes to data structures, and in both cases only a single data structure was changed. Nevertheless, these updates had the highest application times, although it is not possible to determine the impact of single-stepping to the overall cost, because a time breakdown was not provided.

Another unresolved problem with LUCOS' support for updates to data structures is how those data structures are located. LUCOS requires the programmer to provide a function that locates all instances of an affected data structure within the system. This is feasible for data structures that may be referenced in a global array or linked list, but for simple structures located on the heap or stack it is virtually impossible to implement correctly.

LUCOS supports rolling back committed patches by using the original code and data to replace the patched versions. However, rolling back a patch can succeed only if the patch behaved correctly and did not corrupt data structures or lock up the system, so its usefulness and reliability as a recovery tool for bad patches is limited.

The approach taken by LUCOS is based on the implicit assumption that the VMM is more stable and requires fewer updates than the OS. This is probably reasonable in the long term, because a VMM should have a smaller code-base than an OS, however it raises the unresolved question of whether dynamic updates to the VMM are required, and if so how to support them.

Because it only consists of passive modules in the base system, LUCOS' performance overhead is negligible at the expense of higher update costs. However it requires virtualisation, which can have significant overheads depending on the workload and level of IO involved [BDF+03, CG05]. An interesting question to consider is whether a similar system could be implemented that did not require virtualisation. The Xen VMM is used in LUCOS to perform three tasks: halting the system's execution while binary patching is performed, protecting write access to mirrored data structures, and single-stepping execution when those data structures are modified. These functions could be performed within the kernel itself without the need for a virtual-machine monitor. A halt in execution could be achieved by having the update thread disable all interrupts and stop all other processors, entries in the kernel's page table could be marked read-only to cause a page fault when written, and many hardware architectures including x86 support a single-step

mode whereby an exception is delivered after every instruction is executed. Although this design would need to be tested with a prototype implementation, it does not appear that virtualisation is an intrinsic requirement of the LUCOS approach.

The LUCOS system shows promise for updating the Linux kernel, but because there are unresolved problems regarding the performance, practicality and scalability of the design, it remains to be seen whether it is a workable solution to the problem of dynamic updates to operating systems.

2.3.2 DynAMOS

DynAMOS [MR07] is another dynamic update system for Linux. Like LUCOS, it is implemented as a loadable module and avoids any modifications to the kernel's source code, but unlike LUCOS it does not require the use of a VMM.

At its core, DynAMOS updates code at the function level using *adaptive function cloning*, a dynamic instrumentation technique similar to those described in Section 2.2.3. To update a function, the beginning of the original function is replaced with a jump instruction that causes all calls to that function to be redirected to a redirection handler, that performs bookkeeping and decides which version of the function to invoke. To avoid a race condition when the beginning of the original function is overwritten, hardware interrupts are disabled during this operation, limiting DynAMOS to single-processor systems. The authors discuss possible multiprocessor support, but it requires the interruption and synchronisation of all CPUs.

DynAMOS supports limited updates to data structures; fields may be added to existing structures through the use of shadow structures. Shadow structures contain only the added fields, and are located through the use of hash tables that map the address of a structure to the location of its shadow. Update code must explicitly maintain and use shadow structures, and because they are maintained separately from the original data, extra overhead is incurred in their use.

Although DynAMOS can apply some simple updates without achieving quiescence, for many updates it is essential. To detect quiescence, functions that must be quiescent are normally updated to maintain entry and exit counters, however in some cases this approach fails, and the system falls back to walking the stack of every kernel thread, checking for any pointers to the affected functions.

As a loadable module that requires no previous modifications to the underlying kernel, DynAMOS does not impact base performance, however the performance impact of updates is significant. The use of function cloning and execution redirection means that for every updated function, there is additional overhead after an update is applied. Micro-benchmark results show that although this overhead is in the range of 1–8% for long-running functions, it can be much higher, with above 40% overhead incurred for updates

to some functions. The largest update that was tested took 2.30 seconds to complete, but it is not clear for what portion of this time interrupts were disabled, nor the subsequent impact on system throughput.

Like LUCOS, DynAMOS shows promise in its ability to support complex updates to the Linux kernel, however it also has serious unresolved issues regarding its performance and scalability. Because each update incurs a slowdown in system throughput, the applicability of the system is limited and the motivation to reboot as soon as possible after an update is still high.

2.3.3 Ginseng

The authors of Ginseng [NHS+06], a tool for updating C programs discussed previously in Section 2.1.4, are investigating its use for dynamic updates to the Linux kernel [NH06]. They have identified two key problems in applying Ginseng to Linux: First, because Ginseng's static updateability analysis supports only single-threaded programs, it cannot be applied to the kernel, which consists of low-level highly concurrent code. To address this problem, the authors propose to annotate the kernel with transactions and use these to determine when to apply updates.

The second problem relates to Ginseng's use of automatically-inserted indirection. Ginseng normally adds indirection to every type and function, however to do this in the Linux kernel would not only have serious performance implications, it would break correctness for some types where the layout and representation of that type is fixed (for example, page table entries). To address this problem the authors propose annotating types that have a fixed representation or where no change is expected. These would be compiled as-is, whereas types that may be changed would have indirection added to enable future changes in their representation, with the corresponding overheads of indirection. Similarly, functions could configurably be compiled with and without indirection, leading to a trade-off between dynamic patch size and complexity and run-time overhead of function indirection.

As it is work in progress, it remains to be seen whether this approach is tractable, and how it performs in regard to the amount of Linux source code that must be modified, and the runtime overhead of supporting dynamic updates.

2.3.4 AutoPod

AutoPod [PN05, OSS+02] is a virtualisation service within Linux that provides checkpoint, migration and restart of processes transparently to both the application and the kernel. Using AutoPod, it is possible to checkpoint an application on one version of a

kernel, and then restart it on a different updated version of the kernel, as long as the user-kernel interface remains the same and the AutoPod software runs on the new kernel, which is mostly true for maintenance releases.

By combining AutoPod with an underlying virtual-machine monitor it is possible to achieve similar results to dynamic update of the OS, updating the kernel transparently to the applications running on it. This is done by starting a new virtual machine instance with an updated kernel, and then checkpointing the user processes in the old virtual machine before migrating and then restarting them on the updated kernel. Because AutoPod is implemented as a loadable kernel module that intercepts system calls, its overhead is relatively low. The authors report CPU overheads of up to 10% for most system call microbenchmarks, and less than 4% for application workloads.

Microvisor [LSS04] is a VMM specifically optimised for the needs of online maintenance. It avoids IO and memory virtualisation, and is optimised for the case when only one virtual machine is present. It can offer significant performance advantages over traditional virtual machines in an AutoPod-like environment, where multiple virtual machines are used only during upgrades. In the normal case of a single active virtual machine, Microvisor's CPU overhead is negligible; it rises to the comparatively low 5.6% when multiple virtual machines are required.

The combination of the AutoPod and Microvisor designs represents a radically different approach to the problem of operating system updates, and has some compelling advantages. It avoids significant changes to the OS, and does not require any special development model for patches. It suffers some runtime overhead from virtualisation and system call interception, although this is relatively low. However, there are a number of drawbacks to the approach. First, because AutoPod is implemented as a kernel module and accesses kernel data structures to save and restore some process state, it must be maintained and updated to match changes in different kernel versions. For example, in Linux 2.4.14 a field was added to the data structures for TCP connections, and AutoPod had to be modified to support this; in practice, this means that AutoPod would need to be tested with each new kernel version before it could be used for dynamic updates. Second, checkpointing, transferring and restarting applications takes significant time, in the order of 0.5s for a web server and 2s for a desktop environment, and during this time the entire system is inaccessible to its users and will not receive network packets sent to it, although it is still a big improvement over rebooting.

2.4 Conclusions

This literature review has covered two branches of research: dynamic update and operating system modularity and customisability. These fields are separately approaching

problems related to dynamic updates to operating systems, but neither solve all the specific issues that have been identified.

Dynamic update systems generally target application-level software. They fail at updating operating systems code, because they either rely on high-level languages that are not used to implement operating systems, require runtime support that is not available in a kernel environment, use mechanisms that are inappropriate for an operating system kernel, or do not support key requirements of operating systems code such as threading and concurrency. Some domain-specific update systems have been developed that make different design choices appropriate for the constrains of their problem domain. Similarly, rather than attempting to adapt an ill-suited general-purpose dynamic update system, better results may be achieved by designing a dynamic update system specifically for operating systems.

In the area of operating systems research, motivations such as adaptability and extensibility as well as the classic software engineering concerns of reliability and maintainability are pushing modern operating systems to become more modular and componentised. Techniques such as abstract data types, data hiding and encapsulation, and separation of concerns are all features of these modular interfaces. Furthermore, many research projects have investigated making operating system kernels more dynamic, and developed dynamic structures and mechanisms to enable adaptation and customisation. These include replugging and interposition mechanisms, portals, hot swapping, and dynamic kernel instrumentation mechanisms among others. Unlike the problems faced by general-purpose dynamic update systems, these mechanisms are suitable for use in an operating system kernel as a result of their design requirements. However, although some researchers have discussed the possibilities of performing dynamic updates using these mechanisms, they do not on their own provide all the requirements of a dynamic update system.

Finally, a handful of research projects have directly approached the problem of dynamic update for operating systems, however as has been discussed the existing approaches suffer various drawbacks and trade-offs, and it is apparent that there is another point in the design space yet to be explored. That is, the use of dynamic kernel mechanisms from operating systems research, combined with ideas from dynamic update research, to provide a domain-specific dynamic update system for operating systems. This possibility will be explored in the following chapter, where an approach is developed that uses operating system modularity as the basis for a dynamic update system. Later, the results of this approach will be explicitly compared to the other work on dynamic update for operating systems, in Section 8.2.

3 Research Design

From the previous chapter's review of the literature, it is clear that modern operating systems, even those with a monolithic kernel, are becoming modularised and componentised. This modularity is motivated by many advantages including safety, maintainability, extensibility, and configurability. Software construction techniques such as abstract types, data hiding and encapsulation, and separation of concerns are all features of these modular interfaces. Many previous dynamic update systems have used modules or components as the basis of dynamic update [Fab76, HG98, Dug01, BLS+03, ALS06], suggesting that module-based dynamic update for operating systems code may be feasible.

The hypothesis of this dissertation is that dynamic update for operating systems can be achieved through the use of modularity, that by building dynamic update mechanisms around the existing modules within an operating system, and performing a dynamic update as a series of individual modular updates, it will be possible to dynamically update the operating system as a whole. For it to address the motivating problem, such a dynamic update system should support all kinds of changes commonly required for maintenance, including bug fixes and security patches. Furthermore, it should have an acceptably low performance impact, and should not constrain the usability or scalability of the system.

In the remainder of this chapter, a number of fundamental requirements for providing module-based dynamic update in an operating system are developed and explained in Section 3.1. Research methods based around a case-study implementation of modular dynamic update for an experimental operating system are then designed in Section 3.2, before Section 3.3 concludes.

3.1 Requirements for modularity as a basis for dynamic update

The approach being taken is to develop mechanisms that use the existing boundaries within a modular kernel to update the code and data within a module without affecting the rest of the running system, and then to repeatedly apply those mechanisms to update all the modules involved in a larger change, achieving a whole-system update as a series of small self-contained changes. This will require a number of mechanisms for safely updating a specific module, and transforming the data structures maintained by that module.

The fundamental requirements of this approach are now identified. These are updatable units in the form of modules, support for achieving a safe point for performing an update, mechanisms for tracking and transferring state information maintained by updateable units, the ability to redirect invocations when a unit is updated, and a form of version management.

3.1.1 Modularity

In order to update any system, it is necessary to be able to define an updatable unit, which is the smallest part of the system that changes during an update. In this approach, the updatable units of the system will be kernel modules. Depending on the structure of the system, a unit may consist of a code module, or of both code and encapsulated data. In both cases, there must be a clearly defined interface to the module. Furthermore, external code should invoke the module in a well-defined manner, and should not arbitrarily access code or data of that module.

Classic examples of such modular interfaces within an operating system kernel are the Unix virtual file system layer [Kle86] and streams facility [Rit84]. Other examples within the Linux and BSD kernels include the device-driver and network-filter interfaces.

This requirement for encapsulated state and clearly-defined modules is broadly similar to the requirements for microreboots and crash-only software [CKF+04]. An application or operating system that was written to be crash-only would be trivial to dynamically update, due to the segregated state information and isolated modules.

3.1.2 Safe point

To maintain program correctness, dynamic updates must not occur while any affected code or data is being accessed; doing so could cause undefined behaviour. It is therefore important to determine when an update may safely be applied. In general however, this is undecidable [GJB96]. Thus, system support is required to achieve and detect a safe point. Potential solutions involve requiring the system to be programmed with explicit update points, or blocking accesses to a module and detecting when it becomes idle, or *quiescent*.

An operating system is fundamentally event-driven, responding to application requests and hardware events, unlike most applications, which are structured as one or more threads of execution. As discussed later, this event-based model can be used to detect when a module of the system has reached a safe point. Additional techniques can be employed to handle blocking I/O events or long-running daemon threads.

The requirement for modularity plays an important role in achieving quiescence. Without limits on the way in which a module can be invoked, it becomes impossible to detect or prevent invocations, and thus to achieve a safe point. Therefore, a system that is constructed without well-defined modules can only be updated in a single operation.

3.1.3 State tracking

For a dynamic update system to support changes to data structures, which was identified in Section 2.1 as an important requirement for maintenance updates, it must be able to locate and convert all such structures. This requires identifying and managing all instances of state maintained by a module. State tracking can be performed separately for each data structure using ad-hoc mechanisms, but to provide it in a uniform fashion requires functionality often implemented in software systems using the factory design pattern [GHJ+95]. Note that a limited form of dynamic update, changes to code and changes to global single-instance data, is still possible without factories, but it is not possible to support dynamic update affecting multiple-instance data without some kind of state tracking mechanism.

3.1.4 State transfer

When an update is applied that changes data structures, or when an updated module maintains internal state, the state must be converted and transferred, so that the updated module can continue transparently from the module it replaced. The state transfer mechanism performs this task, and supports changes to data structures.

State transfer for a dynamic update system can be as simple as providing the updated module with a pointer to the old module's data, and requiring each update to internally handle or convert the data structures used by the previous version. Alternatively, more general schemes involving common intermediate data representations could be used.

3.1.5 Redirection of invocations

After an update occurs, all future requests affecting the old module must be redirected to the updated version. This includes invocations of code in the module, and accesses to its data structures. The system must provide some mechanism for achieving this redirection, for example by using indirection for module calls, rewriting call points, or by binary patching the code of the old module to redirect execution flow to the new version. Furthermore, in a system supporting multiple-instance data structures, creation of new data structures of the affected type should result in the updated data structure.

3.1.6 Version management

In order to package and apply an update, and in order to debug and understand the running system, it it necessary to know what code is actually executing. If an update depends on another update having previously been applied, then support is required to be able to verify this. Furthermore, if updates are from multiple sources, the versioning may not be linear, causing the inter-dependencies between updates to become complex and difficult to track.

The level of support required for version management is affected by the complexity of update dependencies, but at a minimum it should be possible to track a version number for each update present in the system, and for these version numbers to be checked before an update is applied.

3.2 Research methods

The primary research method used in this dissertation is a case-study implementation of modular dynamic update in the K42 research operating system. K42 is an experimental operating system for 64-bit PowerPC systems primarily developed at IBM Research. It was selected for the initial implementation for a number of reasons:

- K42 is heavily object-oriented, making it suitable for a prototype implementation of dynamic update because the system's pervasive modular structure allows dynamic update features to be developed for all parts of the system more rapidly than on other operating systems that, although modular, lack the consistent module invocation mechanisms used by K42.
- As was discussed in Section 2.2.2, previous work on K42 developed hot-swapping mechanisms that allow code within the kernel to be dynamically changed [AHS+02, SAH+03, AHS+03], and these mechanisms can be re-used for dynamic update, further reducing the complexity of its implementation.
- Despite its role as a research operating system, K42 supports the Linux API and ABI, and is able to run many standard applications and full-system benchmarks, allowing a more thorough performance analysis than would have been possible with other experimental operating systems.
- K42 has been developed over a period of nearly 10 years, and has a rich revision history from which changes could be selected to develop into experimental dynamic updates.

- Finally, during most of the time period that this work was performed, K42 was actively supported and developed by a team at IBM.

Full background information on K42 will be given in Section 4.1. The rest of Chapter 4 describes how K42 fulfils the requirements developed in the previous section, and details the design, implementation and testing of dynamic update in that system.

To test that the system can support the desired maintenance updates, including bug fixes and security patches, two research methods will be used. First, a number of example changes will be selected from K42's revision history and developed into dynamic updates. These changes are selected to cover as many different scenarios as possible; for example, changes to code only, changes to data structures, various changes to module interfaces, and so on. These test updates are described in Section 4.3. Second, to determine whether the system can support all the desired maintenance updates, an analysis of the complete K42 CVS revision history will be conducted. This analysis, which is described in Chapter 5, will examine changes in the revision history to decide if they are suitable for application as dynamic updates.

To measure the performance and scalability impact of the dynamic update system, a series of experiments will be performed, as described in Chapter 6. These include measurements of the added performance overhead required for dynamic update support, as well as the direct costs incurred during a dynamic update. Finally, the design used in the K42 case study will be generalised to support for other commodity operating systems, such as Linux and FreeBSD, through a discussion in Chapter 7.

3.3 Conclusions

In this chapter, an approach for dynamic update based on operating system modularity has been outlined. Fundamental requirements for this approach have been developed; namely modularity, support for detecting a safe point, state tracking and state transfer mechanisms, support for redirection of module invocations and support for version management. Finally, a series of research methods for evaluating the approach have been selected, based on a case-study implementation of dynamic update in the K42 research operating system. In the following four chapters, each different aspect of the research plan will be developed and the results presented, beginning in the next chapter with the design, implementation and testing of a dynamic update system for K42.

4 Dynamic Update in K42

The previous chapter motivated and described an approach for dynamic update based on OS modularity, and outlined a series of research methods based on a case-study implementation of dynamic update for the K42 research operating system. This chapter details the design, implementation and testing of that prototype. First, the K42 system is described and necessary background information presented in Section 4.1. Next, Section 4.2 covers the implementation of dynamic update in K42, including a discussion of how K42 provides the fundamental requirements identified in the previous chapter. Finally, Section 4.3 describes a series of updates that were developed to test the system, and Section 4.4 concludes.

4.1 The K42 operating system

K42 is an experimental operating system developed at IBM Research [IBM02a, AAB+05, KAR+06]. K42 targets 64-bit cache-coherent NUMA systems; at present, it runs on PowerPC hardware, although in the past the MIPS architecture was supported, and a port to x86-64 is in progress. Key goals of the project included scalability for large multiprocessor machines with non-uniform memory access (NUMA), complete compatibility with the Linux API and ABI, and customisability and adaptability of the system's behaviour. To achieve these goals, the designers:

- structured the system using modular, object-oriented code;
- avoided centralised code-paths, global data structures, and global locks;
- moved system functionality from the kernel to server processes and into application libraries.

4.1.1 K42 system structure

K42 is structured around a client-server model, as show in Figure 4.1. Many system services are provided by server processes and are invoked using an IPC mechanism.

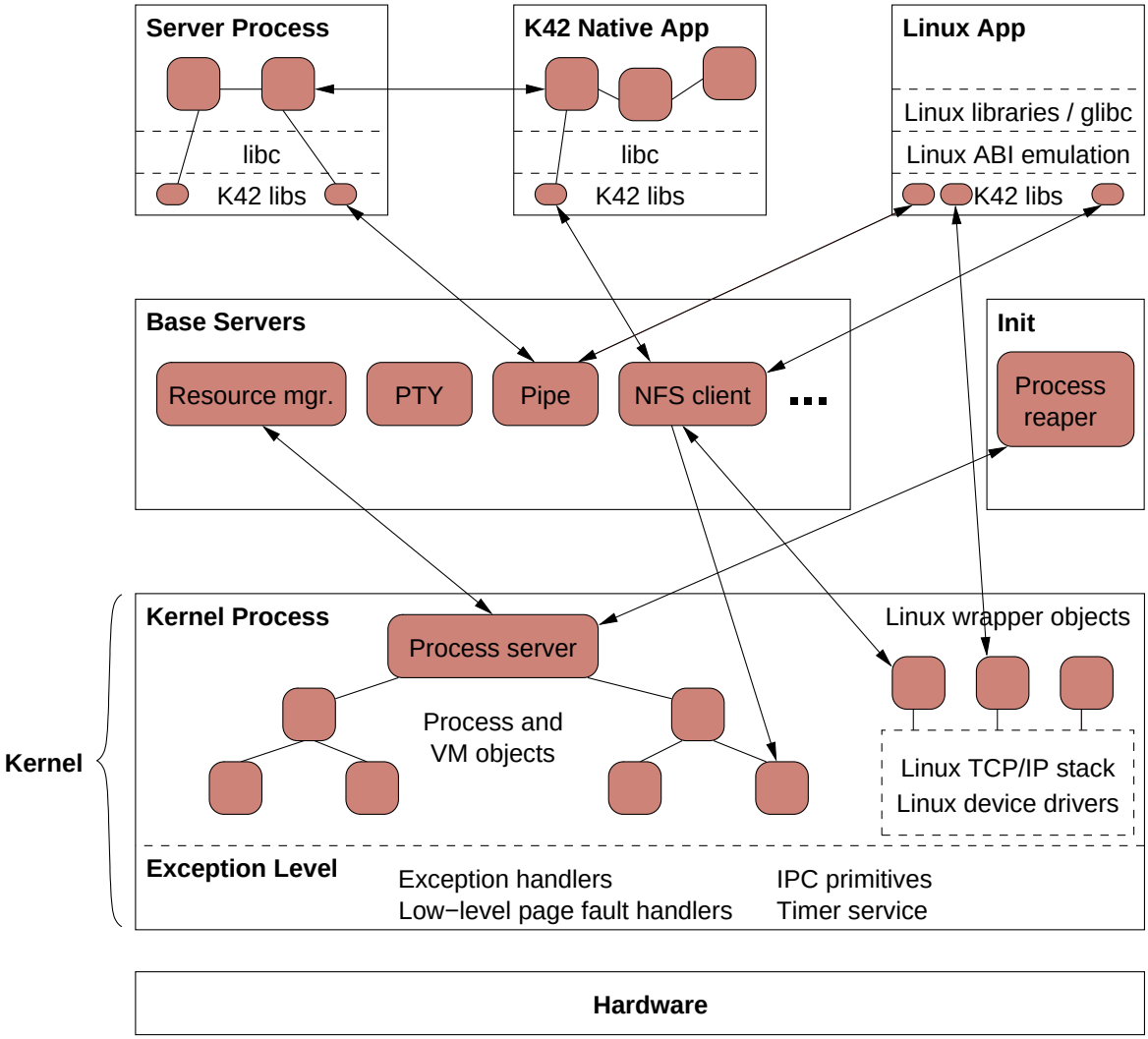

Figure 4.1: Structural overview of K42. Arrows indicate possible protected procedure calls between objects in different processes.

In the kernel, the code at *exception level* provides some of the features commonly found in a microkernel. It implements the lowest-level of exception and page-fault handling, IPC primitives, base scheduling operations, and the timer service. This code runs with interrupts disabled and accesses only pinned data structures. The rest of the kernel, known as the *kernel process*, is threaded, fully pre-emptable, and paged (only its code and some data structures are pinned in memory). Besides running in privileged mode, the kernel process is similar to a user-level process, because its services are invoked using the IPC mechanisms implemented at exception level. The main services provided by the kernel process are process management and the full virtual memory system. K42's device drivers and network stack are reused from Linux and also run in the kernel process [IBM02c].

Remaining system services, including the resource management policy, file systems, socket and pipe communication, pseudo-terminal management, and others are implemented at user-mode, and invoked via the same IPC mechanisms used to invoke kernel process services. Earlier versions of K42 ran these servers as separate processes; however, as Figure 4.1 shows, most are now co-located in a *base-servers* process, primarily to improve performance of the fork() operation. The base-servers process is the first user process started at boot time, and has access to privileged operations provided by the kernel process that are not accessible to the other user processes.

Communication between clients and servers, including the kernel process, is performed via K42's IPC mechanism, known as a protected procedure call (PPC) [GKS94]. A *stub compiler* is used to automatically generate the code for PPC calls from decorations on the C++ class declarations of server objects. The generated stubs marshal and unmarshal method arguments, and also implement a capability-like security mechanism using the sender identity that is provided by the kernel on message delivery. Thus, C++ classes form the native interface to K42 system services. As much system functionality as possible is implemented in these application-level class libraries. For example, the IO model provided by the kernel and server processes is purely event-driven; blocking operations, such as select(), are implemented within libraries, avoiding unnecessary communication.

An application may be written to use the native K42 class interfaces, or it may be an unmodified Linux application [AAE+03]. In this case, attempts to perform Linux system calls are intercepted via trap reflection, and an emulation library within the process converts the calls into operations on the K42 libraries, that in turn invoke the kernel or servers via PPCs as required. It is also possible for a Linux application to be linked with a modified version of the glibc library that invokes the emulation layer directly, avoiding the overhead of trap reflection. Examples of unmodified Linux applications that have been run on K42 include the OpenSSH daemon and IBM's DB2 database.

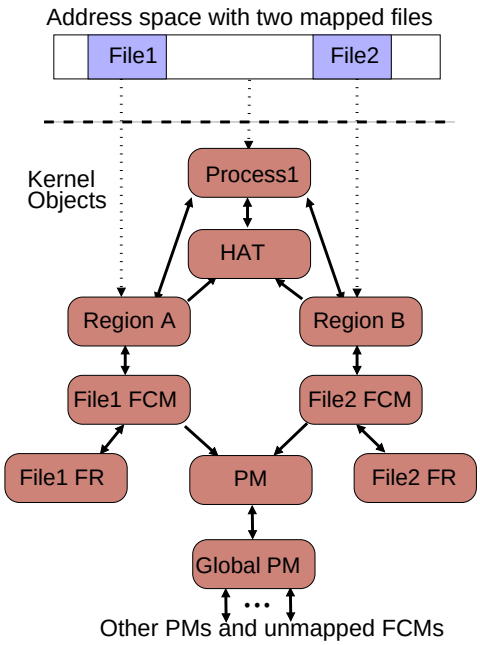

Figure 4.2: Virtual memory system object hierarchy. Reproduced from Appavoo [App05].

4.1.2 Object-orientation and modularity in K42

All of the native (non-Linux) code in K42, including the kernel process, servers, and libraries, is implemented in object-oriented C++. Every resource in the system (for example, a memory region, network socket, or process) is managed by a different set of object instances.

Figure 4.2 shows an example of the object hierarchy used in the virtual memory system [IBM02b]. *Process objects* (analogous to the process control block structure present in other operating systems) represent a process, and maintain a list of memory regions, represented by *region objects*, that make up its address space. Each region is backed by a *file cache manager* (FCM) object, that is responsible for mapping page frames to file data and implementing the local paging policy for each file. Linked to each FCM is a *file representative* (FR) object, whose task is to communicate with the user-level clients of a file, such as the file-system server. Other objects involved in virtual memory are the *hardware address translator* (HAT), which manages the hardware-specific representation of an address space, and the *page managers* (PM), which implement global aspects of the paging policy, such as the allocation of frames to FCMs.

The primary reasons for the use of such fine-grained object decomposition in K42 are scalability and adaptability. Each object contains the meta-data necessary to manage its resource, as well as all the locks necessary to manipulate it. Global locks, data structures, and policies can thus be avoided. In the virtual memory system, the fine-grained object structure allows page faults in separate processes to separate files to be handled entirely in parallel without any lock or data contention. Furthermore, the system can adapt on a per-resource basis, because a different FCM implementation may be used for any open file, for example to implement different caching and pre-fetching policies.

4.1.3 Clustered objects

Although a fine-grained object-oriented design helps scalability, some objects, such as the FCM for a widely-shared file, may still be accessed by many nodes in a large multiprocessor. To address the scalability problem this causes, K42 uses *clustered objects* [GKA+99, App05], a technology that allows objects to implement and control their own cross-processor distribution transparently to their clients.

Figure 4.3 abstractly illustrates a clustered object implementing a simple counter, with operations to increment, decrement and return the value of the counter. Externally, a single instance of the counter object is visible, but internally to the object three different counter values are maintained, one for each processor in the system. Calls on the object are transparently directed to the local processor's counter, allowing the increment and decrement operations to be performed without any cross-processor communication, because they only modify a local value. Only the operation to return the current value of the counter must access the non-local data, resulting in performance and scalability benefits when such operations are relatively infrequent.

Clustered objects are widely used in K42 to improve the scalability of objects that are accessed on multiple processors. For instance, K42 has an implementation of the FCM object that is internally partitioned, allowing requests on different processors to be handled in parallel without unnecessary sharing. This improves the scalability of the system when a single file, such as the C standard library, is widely shared among processes on multiple processors.

One drawback of partitioned clustered objects such as the FCM and abstract counter is that they incur additional time and space overhead when they are only accessed on a single CPU. However, the object-oriented nature of K42 enables adaptability that avoids this problem, because different resources can be managed by different implementations. For example, two implementations of the process object exist, ProcessReplicated, the default, and ProcessShared, which is optimised for the case when a process exists on only a single CPU [AADS+03]. K42 defaults to creating replicated processes, but allows for a combination of replicated and shared processes.

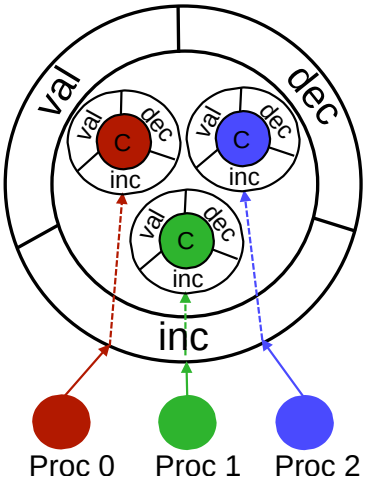

Figure 4.3: Abstract view of clustered object. Reproduced from Appavoo [App05].

The K42 clustered object implementation extends the standard C++ object structure provided by the compiler, as illustrated in Figure 4.4. First, all clustered objects are accessed indirectly via a a global table of object pointers known as the *object translation table*. This table is located in *processor-specific memory*, a region of memory where the same virtual address range maps to different physical memory on each processor, allowing object invocations to be transparently directed to per-processor objects. Second, all methods of clustered objects are declared as *virtual*, causing the compiler to invoke them via a pointer stored in the object's *virtual function table*. This allows different object instances to transparently use different implementations, and is also used to instantiate objects as they are invoked on each processor.

The fine-grained object orientation and clustered objects used in K42 come at a cost, due to the extra indirection required for object invocations, virtual functions, and the object translation table. Nevertheless, the K42 designers chose to structure the system in this way, because the extra indirection enables performance improvements not otherwise possible, through customising objects to application behaviour, and improved locality and scalability on a multiprocessor [IBM02a]. Scalability comparisons with Linux support this decision, with K42 scaling significantly better than Linux 2.4.19 on more than ten processors [AAE+03].

4.1.4 Interposition and hot swapping

Using the indirection provided by the object translation table, the K42 project has developed *object interposition* and *hot-swapping* services [HAW+01, AHS+02]. Interposition

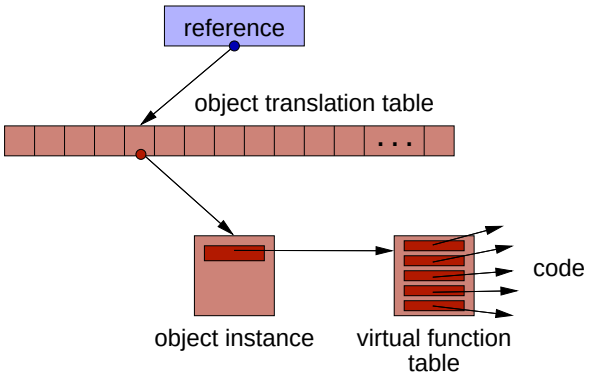

Figure 4.4: Clustered object indirection through object translation table

allows the calls to a clustered object instance to be monitored or redirected. Hot swapping uses interposition, and allows an object instance's entire implementation to be changed at runtime. Hot-swapping technology has been used in K42 to achieve online reconfiguration [SAH+03, AHS+03], for example dynamically switching a caching algorithm from LRU to FIFO when a particular access pattern is detected.

To perform an interposition, the object translation table entry is updated to refer to an *interposer object,* rather than the original object, forcing all calls to the original object to be transparently redirected to the interposer. The interposer object holds a reference to the original object, and may forward calls to this object as well as to wrapper code. Besides its role in the hot-swapping implementation, interposition has other uses in an object-based system; it allows dynamic monitoring of a particular object's performance and invocation patterns, and may be used to determine when to perform an adaptive hot-swap.

Hot-swap mechanism

For an object to be safely hot-swapped, it is necessary to ensure that it is *quiescent*, or that no threads hold references to its data structures, during the critical period when its state is transferred and the new object takes over. To detect a quiescent state in an object, K42 makes use of a *thread generation count* mechanism, which tracks the lifetime of the system's threads, and is similar to read-copy update (RCU) in Linux [MSA+02, MS98]. Logically, each thread in K42 belongs to a certain epoch, or *generation*, which was the active generation when it was created. A count is maintained of the number of live threads in each generation, and by advancing the generation and waiting for the previous generations' counters to reach zero, it is possible to determine when all threads that existed on a processor at a specific time have terminated [GKA+99]. This is feasible

in K42, since the system is event-driven and threads are generally short-lived and non-blocking. Mechanisms also exist to support applications that require blocking threads. The same approach can also be used in operating systems that may have blocking threads but are still event-driven [MSA+02].

In order to perform a hot swap, K42 interposes a hot-swap *mediator object* in front of the object to be swapped out. As illustrated in Figure 4.5, the mediator then passes through several phases: forward, block, and transfer.

Forward In the forward phase, the mediator increases the thread generation count and then continues to forward and track requests to the object until it is certain that any requests started before the hot-swap began have completed, and thus that all existing requests are being tracked. This is achieved by waiting for the count of threads in previous generations to reach zero.

Block Once all calls to the object are tracked, the mediator enters the block phase. In this phase all new incoming calls are blocked, while the existing calls that were being tracked complete. An exception is made for recursive calls, that must be allowed to complete to avoid deadlock. Once all calls have completed or are blocked, the object is known to be in a quiescent state.

Transfer The object is quiescent and can now safely be swapped. To do this, the mediator performs the transfer of state from the old to the new object. State transfer is the process by which an object's data structures are converted to the format required by the new implementation, and is discussed in the following section.

Complete Once the state transfer is complete, the mediator updates the entry in the object translation table to point directly to the new object instead of itself, and then unblocks the calls it was tracking, forwarding them to the new object. The new object is now active, and the mediator and the old object are destroyed.

State transfer

The *state-transfer mechanism* is responsible for converting the data structures used by the old object into the format expected by the new implementation. As was discussed in Section 2.1, state transfer happens also in dynamic-update systems, however in these the replacement module is usually a newer version of the existing code, or has been explicitly designed to replace the existing code, so it is reasonable to assume that the state data is in the same or similar format, and that the replacement module can implement a transfer function that understands the internal data format of the module being replaced.

When hot swapping however, there could be any number of different implementations of a particular service, so it is much more important to be able to negotiate a common

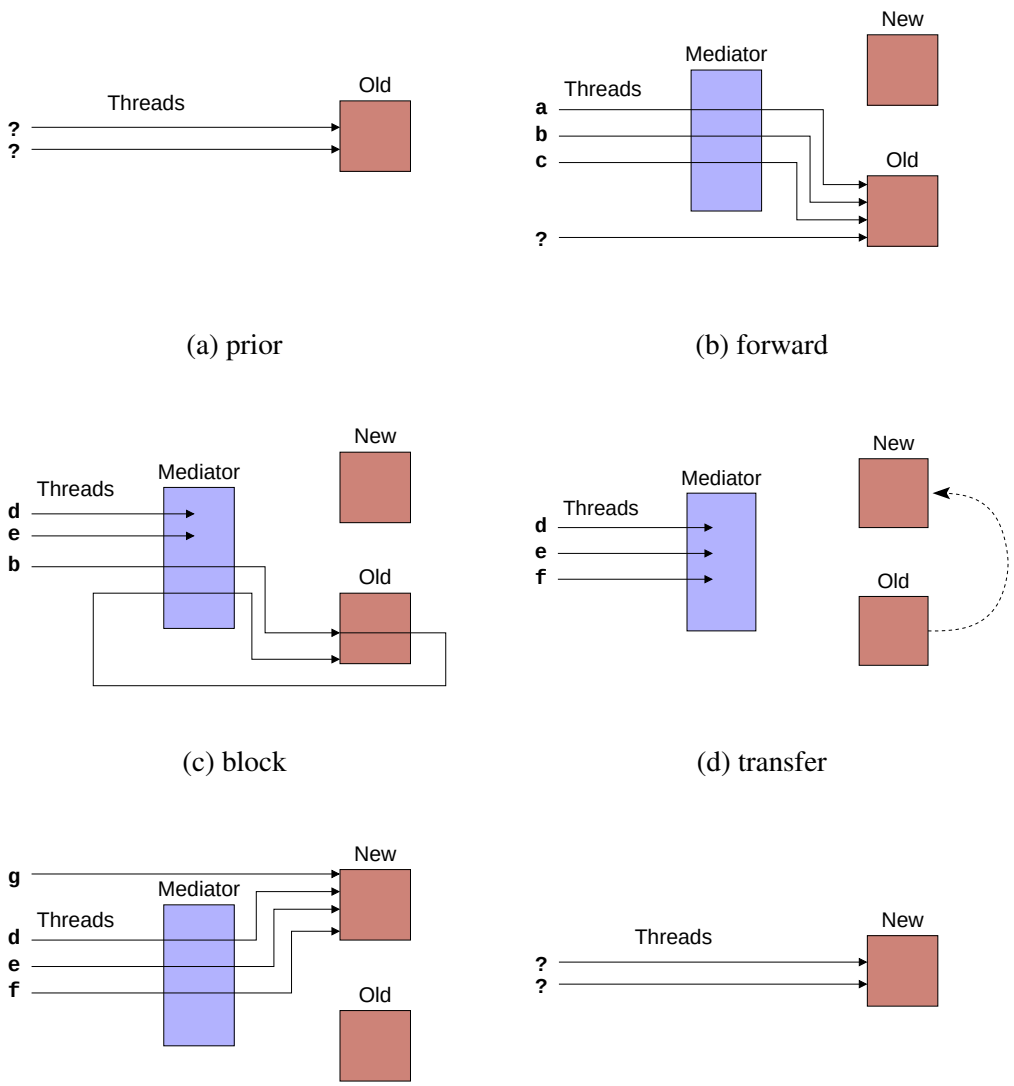

Figure 4.5: Phases in an object hot-swap: (a) prior to the swap threads invoke an object directly; (b) a mediator object is interposed, forwarding new calls to the object; (c) once all calls are tracked, the mediator blocks incoming calls and waits for the existing ones to finish; (d) when the object is quiescent, state is transferred to its replacement; (e) the mediator forwards blocked calls to the new object; (f) the old object and mediator are destroyed. Reproduced from Appavoo *et al.* [AHS+02].

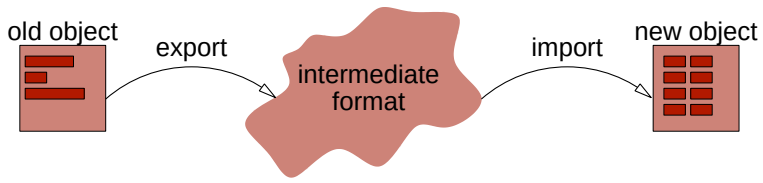

Figure 4.6: State transfer

intermediate format for the transfer of state data. K42 objects therefore export a list of intermediate formats that they support. As shown by Figure 4.6, the mediator performing a hot-swap selects a common format supported by both objects to perform the conversion, before using the export and import functions of the objects to perform the transfer. This approach places the majority of the work involved in state transfer onto the programmer implementing clustered objects, who must choose and implement the appropriate intermediate formats and transfer functions.

4.1.5 Comparison to commodity operating system kernels

As discussed in Section 3.2, K42 was chosen for the case-study implementation for a number of reasons, including its consistent modularity and existing use of clustered objects. However, as this section has shown, K42's structure differs in a number of significant areas from the commodity operating systems that are the aim of this research. Specifically, although commodity operating systems are modularised, the module structure is not as pervasive, and module invocation mechanisms are not as consistent as for K42's clustered objects. Furthermore, K42 always encapsulates data within object interfaces, whereas in commodity kernels some data structures may be accessed directly by code outside a module.

On the other hand, K42 has many features in common with commodity systems. Its API is Linux-compatible, so as a system it implements the same functionality as a commodity kernel. As was described in Section 4.1.1, although some functionality is implemented in user-level servers, many services and policies reside within the kernel, unlike microkernel-based systems. Finally, although it is somewhat unique in its design, K42 is an existing system with many years of development, and dynamic update is a new feature in its evolution.

K42's most important advantage for this case-study over a commodity system such as Linux is not its modularity in and of itself, but more specifically its consistent module invocation mechanisms. These make it appealing for a case study, because dynamic update features can be applied to large parts of the system with minimum implementation effort, whereas on another system such as Linux much more implementation work would

be required to enable dynamic updates to different module interfaces. The implications of these structural differences will be discussed in greater depth in Chapter 7, where the design is discussed in the context of commodity systems such as Linux and FreeBSD.

4.2 Dynamic update implementation

Because it allows the implementation of kernel objects to be changed on-the-fly, hot swapping provides a solid base on which to build dynamic update in K42. However, as was discussed in Section 2.2.2, although hot swapping might be used in a dynamic update implementation, it does not solve the whole problem. Hot swapping only changes the implementation of a specific object instance, rather than all objects affected by a code update, and it does not allow the object's interface to change. Furthermore, K42 only supports hot-swaps to pre-existing object implementations that are present at compilation.

This section details the design and implementation of a dynamic update system for the K42 kernel, based on hot swapping. The design aims to achieve a whole-system dynamic update as a series of coordinated hot-swap operations. This section begins with a discussion of how the previously-identified requirements can be provided in K42, and continues with details of the specific functionality that was added to implement dynamic update, finishing in Section 4.2.6 with a description of the process of preparing and applying a dynamic update.

4.2.1 Requirements

In Section 3.1, the fundamental requirements for a system to support module-based dynamic update were identified. These requirements are now revisited, with a discussion of how they are already provided by K42, or how K42 will be extended to provide them.

Modularity

A clear choice for the dynamically-updatable unit in K42 is the same used for hot swapping, namely the clustered object instance. As discussed, the K42 kernel is structured as a set of objects, and the coding style used enforces encapsulation of data within objects. Each object's interface is declared in a virtual base class, allowing clients of an object to use any implementation, and for the implementation to be changed transparently by hot swapping.

Safe point

As detailed in the previous section, the implementation of hot swapping first blocks new invocations of an object, and then uses the thread generation count to detect quiescence, thereby achieving a safe point for an update.

State tracking

State tracking is a requirement that K42 lacks. State in K42 consists of clustered object instances, however these instances are tracked in an ad-hoc, class-specific manner, and are usually created through calls to statically-bound methods. To address this problem, factory objects are introduced in K42; these are responsible for object creation and destruction, track object instances in a uniform fashion, and are described in detail in Section 4.2.3.

State transfer

As was discussed in Section 4.1.4, K42 performs state transfer via conversion functions to and from a common intermediate representation. This generalised technique was developed to support hot-swapping between arbitrary implementations of an interface. However, in the case of dynamic update, the replacement object is usually a slightly-modified version of the original object, with a similar internal state representation, so the conversion functions in these cases perform either a direct copy, or a copy with slight modifications.

In cases where an object's state is large, or when many object instances must be updated, a copy is an unnecessary expense, because the updated object is deleted immediately afterwards. For example, the process object maintains a series of structures which describe the address-space layout. To avoid the cost of deep-copying and then discarding these structures, the data-transfer functions involved simply copy the pointer to the structure and set a flag in the old object. When the object is destroyed it checks this flag and, if it is set, does not attempt destruction of the transferred data structures. Effectively, ownership of the structure is transferred to the new object instance. This only works in cases where the new object uses the same internal data format as the old object, which is true in many dynamic update situations. In cases where this is not true, the negotiation protocol ensures that a different transfer function is used.

Redirection of invocations

K42's object translation table is ideal for redirecting object invocations after a hot swap or dynamic update. In the process of performing a dynamic update, the translation table

entries for an object are updated to point to the new instance, causing future calls from clients to transparently invoke the new code. The object translation table was originally designed in K42 to support the clustered object system, but has also been utilised to implement hot swapping and dynamic update.

When an object that has multiple instances is updated, creations of that type must also be redirected. This redirection is provided by the factory mechanism, described in Section 4.2.3.

Version management

Version management in K42 was lacking, because there had been no previous work on dynamic update, and thus no consideration of the need to support multiple versions of objects. To address this limitation, a simple version scheme was developed for dynamic updates in K42. Each factory object carries a version number, and before any update proceeds these version numbers are checked for compatibility. Further details follow in Section 4.2.6.

4.2.2 Module loader

To perform updates, the code for the updated object must be present. The normal process for adding an object to K42 was to recompile the kernel, incorporating the new object, and then reboot the system. This is insufficient for dynamic update, so a *kernel module loader* that is able to load the code for an updated object into a running kernel or system server was developed.[1]

A K42 kernel module is a relocatable ELF file with unresolved references to standard kernel symbols and library routines. The module loader consists of a special object in the kernel that allocates pinned memory in the kernel's text area, and a trusted user-space program that has access to the kernel's symbol table. This program uses that symbol table to resolve the undefined symbols in the module, and load it into the special region of memory provided by the kernel object. It then instructs the kernel to execute the module's initialisation code.

The K42 module loader operates similarly to that used in Linux [dGdS99, BC02], but is simpler. Whereas Linux must maintain a dynamic symbol table and support inter-dependencies between modules, in K42 this is avoided because all objects are invoked indirectly through the object translation table. A module can (and to be useful should) contain code that is called by the existing kernel without requiring its symbols to be visible. Its initialisation code instantiates replacement objects and performs hot-swap or

[1] The K42 module loader was developed for this work by Jeremy Kerr.

dynamic update operations to invoke the code in those object instances. The module loader performs relocation and symbol-table management at user-level, leaving only the space allocator object in the kernel.

4.2.3 Factory mechanism

Hot swapping allows updates to the code and data of a single specific object instance. However, K42 is structured such that each instance of a resource is managed by a different instance of a class. To dynamically update a kernel class, the infrastructure must be able to both locate and hot-swap all objects of that class, and cause any new instantiations to use the updated code. Note that, as detailed in Section 2.1, this problem is not unique to K42; to support dynamic updates affecting data structures requires a mechanism to track all instances of those data structures and update them.

Previously in K42, object instances were tracked in a class-specific manner, and objects were usually created through calls to statically-bound methods. For example, to create an instance of the ProcessReplicated class (the implementation used by default for process objects), the following call was used:

```
ProcessReplicated::Create(ProcessRef &out, HATRef h, PMRef pm,
                          ProcessRef creator, const char *name);
```

This leads to problems for dynamic update, because the Create() call is bound statically at compile-time, and cannot easily be redirected to an updated implementation of the ProcessReplicated class, and also because the caller of this method is relied upon to track the newly-created instance.

To track object instances and control object instantiations, the factory design pattern [GHJ+95] was used. In this design pattern, a factory method is an abstraction for creating object instances. In contrast to the original intent of the design pattern, where a factory method serves to abstract the exact class of object that will be created, the primary purpose of factory objects in the implementation of dynamic update for K42 is to track object instances in a unified manner. The term *factory* is, therefore, being used more generally to refer to a level of indirection on object creation, rather than the specific purpose of the original factory method design pattern on which it is based.

As well as creating instances, factories in K42 also record and track those instances, and are themselves clustered objects. As shown in Figure 4.7, each factory is accessed through a well-known factory reference, provides an interface for creating and destroying objects of one particular class, and maintains the set of references to all live objects that it has created.

The majority of the factory implementation is factored out using inheritance and preprocessor macros, so that adding factory support to a class is relatively simple. Using the previous example, after adding the factory, the creation call was changed to:

```
DREF_FACTORY_DEFAULT(ProcessReplicated)->create(...);
```

where `(...)` represents the arguments as before. The macro above hides some implementation details, whereby the default factory for a class is accessed using a static member that holds the factory reference; it expands to the following:

```
(*ProcessReplicated::Factory::factoryRef)->create(...);
```

Making the factory a clustered object and accessing it through an indirect reference allows the factory itself to be hot-swapped and updated, and is used during the process of applying a dynamic update, as will be described in Section 4.2.6.

To provide rudimentary support for configuration management, factories carry a version number identifying the specific implementation of the factory and its type. The factories in the base system all carry version zero, and updated factories have unique non-zero version numbers. A strictly linear model of update is assumed. When an update occurs, the current version number of the factory is compared to the version number of the update, and if the update is not the immediately succeeding version number, the update is aborted. To support reverting updates in this scheme, the previous version could be reapplied with an increased version number.

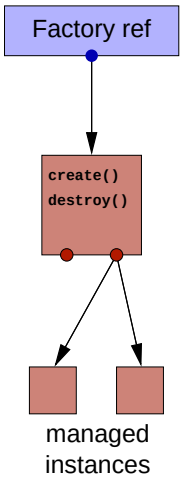

Figure 4.7: Factory object

Performance and scalability influenced the implementation of the factories. For example, object instances are tracked for dynamic update in a distributed fashion using per-CPU instance lists.

It was found that adding factories to K42 was a natural extension of the object model, and led to other advantages besides dynamic update. As an example, in order to choose between ProcessReplicated and ProcessShared, K42 had been using a configuration flag that was consulted by the code that creates process objects to determine which implementation to use. Using the factory model, this flag could be removed, allowing the scheme to support an arbitrary number of implementations, by changing the default process factory reference to the appropriate factory object.

4.2.4 Lazy update

An early implementation of dynamic update 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 [NHS+06, CCZ+06]. 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, the system performance was severely degraded while converting all the affected objects.

To illustrate the scale of this problem, and show that it is not unique to K42, the /proc/slabinfo file was used to count instances of different kernel data structures on a moderately-loaded file and compute server running Linux 2.6.18. To take a few examples, 1.9 million each of the file-system's inode and vnode structures, 234,264 blocks in the buffer cache, 51,301 virtual memory areas, and 14,437 open files were found. 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, the ability was added to perform updates lazily [BLS+03]. When a lazy update is loaded, affected object references are changed to point to a special lazy-update object. The first time such an object is invoked, it initiates the actual update, removes itself from the affected object reference, and then restarts the method call that triggered it.

The lazy update object was implemented as an *arbiter*. Arbiters were developed in recent work by Raymond Fingas [Fin06], and provide a general framework for interposing on and modifying calls to clustered objects in K42. The arbiter implementation developed by Fingas was modified in two ways to support lazy update. First, a special case was added to the method return path to allow a call that triggered a dynamic update to be restarted on the newly-updated object. Second, to enable a more efficient implementation, arbiters were changed to allow a single arbiter object to interpose on multiple target objects.

During the period that object references are being marked for lazy update, it is possible that both old and new versions of an object are active together, however this is not a problem, because even objects of the same class must only access each other through calls on their clustered object references. The same situation occurs in the regular eager implementation of dynamic update, which updates object instances concurrently with other system activity.

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 the system to avoid unnecessarily converting objects that are not invoked between updates or ever again. If an object instance has been only lazily updated, and another update to that object is loaded, the state-transfer functions from both update versions could be used 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.

4.2.5 Adaptor objects

Hot-swapping does not support changes to object interfaces. 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. 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 lock-up or crash).

From an examination of the changes to interfaces observed in the K42 revision history, which will be covered in Chapter 5, it was found that most interface changes, and all of those involved in bug fixes, 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.

These observations led to the use of the object adaptor design pattern [GHJ+95]. Adaptors wrap a class to make it provide a different interface. In K42, adaptors are implemented as arbiter objects [Fin06], and act transparently to callers 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;

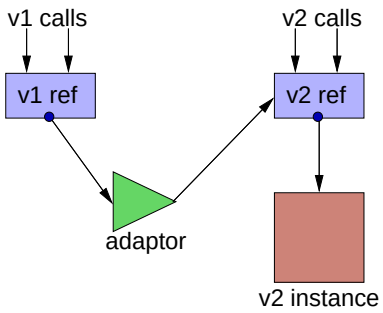

Figure 4.8: Adaptor object

- 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. Approaches for handling such interface changes will be discussed in Section 9.1.3, however, the forms of interface change supported by adaptors are sufficient for maintenance changes, as will be shown in Chapter 5.

The design of an adaptor is shown in Figure 4.8. 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. A special method on the old reference allows caller objects to discover the new reference when they themselves are updated. Once all old caller objects are updated to use the new interface, 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, and the places at which it calls into higher-level code are limited.

The use of adaptor objects follows the fundamental design principle of applying dynamic updates as a series of small independent changes. Adaptors allow the system to be updated 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, the adaptors are removed.

4.2.6 Steps in a dynamic update

Factories are used to implement dynamic update in K42. Kernel developers implement changes as normal, providing a new version of one or more classes. To build a dynamic update, an update developer must 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. To apply the update, the module is loaded into the kernel, and its initialisation code is automatically executed. As shown in Figure 4.9, the initialisation code triggers the following steps:

1. A factory for the updated class is instantiated. At this point, the version number of the updated factory is checked against the version number of the existing factory, and if it is incorrect the update is aborted.

2. The old factory object is located using its statically-bound reference, and hot-swapped to the new factory object; during this process new factory receives the set of instances that was being maintained by the old factory. This is performed by the state-transfer functions, that are shown in Figure 4.10.

3. Once the factory hot-swap has completed, the old factory is destroyed. At this point, due to the factory reference having been changed by the hot-swap, all new object instantiations are being handled by the new updated factory, and therefore go to the updated class. However, any old instances of the object have not yet been updated.

4. To update the old instances, the new factory traverses the set of instances it received from the old factory, and performs the following tasks for each:

    a) For each old instance it creates an instance of the updated object, and initiates a hot-swap between the old and the new instances. Alternatively, if lazy update is being used, it only marks the instance 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 instances of the class, use lazy update.

    b) 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

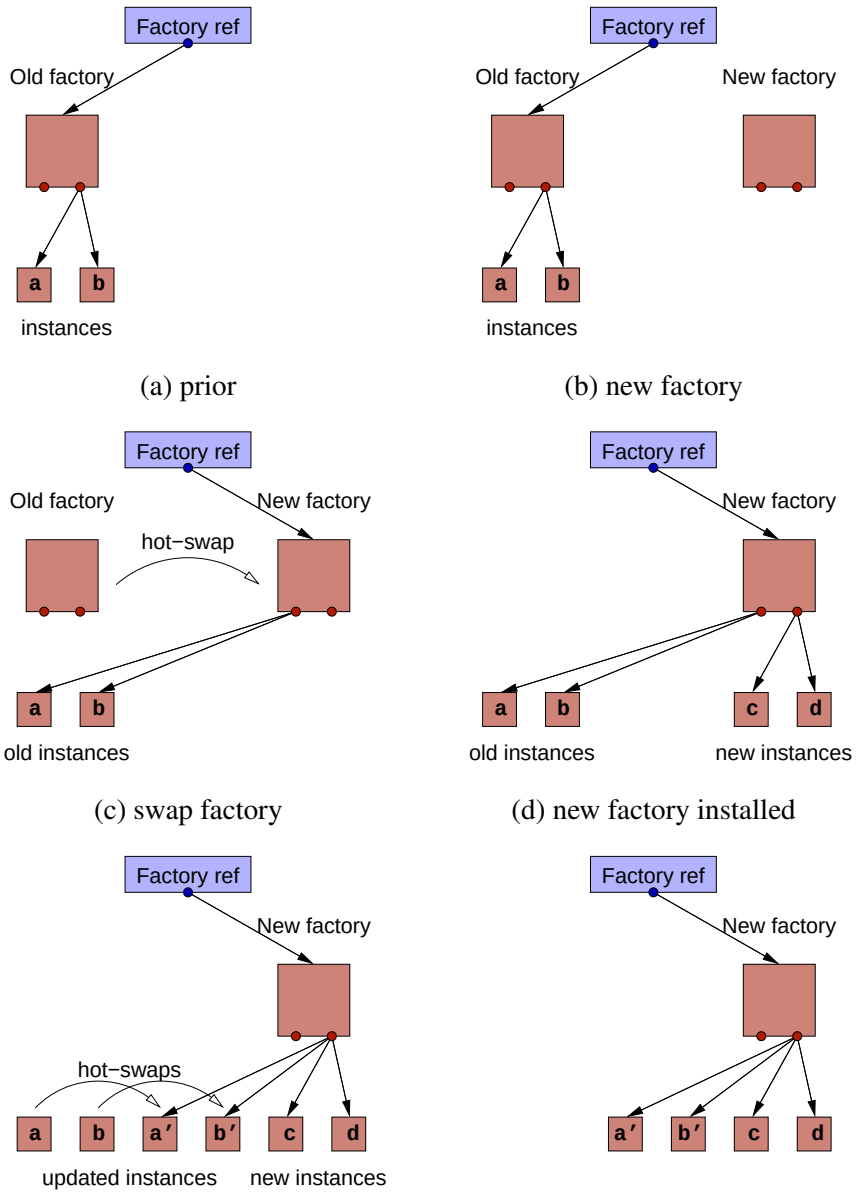

Figure 4.9: Phases in the dynamic update of a multiple-instance object without adaptors or laziness: (a) prior to update the old factory is maintaining instances of a class; (b) instantiate a new factory for the updated class; (c) hot-swap the factory with its replacement (transferring the set of managed instances, and transparently swinging the factory reference); (d) after the hot-swap, new instantiations are handled by the updated factory code (thus creating objects of the new type); (e) update old instances by hot swapping each to an updated replacement; (f) the update is complete.

```
DataTransferObject *BaseFactory::Root:: dataTransferExport (DTType dtt, VPSet
    transferVPSet) {
    tassert (dtt == DTT_FACTORY_DEFAULT, err_printf("wrong transfer type\n"));

    /* get list of old instances */
    BaseFactory *rep = (BaseFactory *)getRepOnThisVP();
    InstanceList * instanceList = rep->oldInstanceList;
    rep->oldInstanceList = NULL;
    if (instanceList == NULL)
        instanceList = new InstanceList ();

    /* create combined instance list by merging old and new lists */
    rep->instanceList.locked_transferTo (* instanceList ); // no need for locks when
        quiescent

    /* create transfer object containing instance list pointer to be passed to the
        import function */
    return new DataTransferObj( instanceList , (FactoryRef)getRef() );
}

SysStatus BaseFactory::Root:: dataTransferImport (DataTransferObject *dtobj, DTType dtt,
    VPSet transferVPSet) {
    tassert (dtt == DTT_FACTORY_DEFAULT, err_printf("wrong transfer type\n"));

    /* copy instance list pointer to our old instance list */
    BaseFactory *rep = (BaseFactory *)getRepOnThisVP();
    rep->oldInstanceList = ((DataTransferObj *)dtobj)->getInstanceList();

    /* delete transfer object */
    delete dtobj;

    return 0;
}
```

Figure 4.10: Simplified state-transfer code for factory objects.

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, whether or not its own interface changes, the new reference must be located for any objects it holds references to. The object's state-transfer function does this by using a special method implemented in the base classes for all objects, that returns the canonical reference for a given object, bypassing any adaptor.

c) As object updates complete, either directly or lazily, the old objects and lazy-update objects are destroyed.

This step proceeds in parallel across all CPUs where the old factory was in use, and while the rest of the system is functioning. Because each object instance is hot-swapped individually, and because K42 encapsulates all data behind object instances, there is no requirement to block all accesses to all objects of the affected type while an update is in progress.

5. When all old objects of a given type are updated, as determined by the relevant factory, two cleanup operations occur. First, any adaptor objects for classes called by the updated class 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, the current module loader implementation does not yet free memory.

In some special cases, for example when an update adds new objects to the system that do not replace any existing objects, or when an update affects an object with only a single instance, the full dynamic update implementation is not required. In these cases the initialisation code in the module is simpler; for example, it may simply register the new class, or directly hot-swap the single instance of the old object.

4.3 Testing dynamic update

In the course of developing the dynamic update mechanisms described in the previous section, several example updates were prepared as test cases. These examples were selected from actual developer changes in K42's revision history, and all relate to the memory management code of the K42 kernel [IBM02b]. They are described here, in order of increasing complexity, starting from an update that is a simple loadable module, and ending with a change that requires both factory objects and an adaptor. Some of these examples will be revisited in Chapter 6, where they are used in performance measurements.

4.3.1 New kernel interfaces for partitioned memory region

Benchmarking of a memory-intensive parallel application had showed poor scalability during its initialisation phase. Analysis of the problem had determined that a bottleneck occurred during the resizing of a shared hash table structure, and a new partitioned memory management object had been developed that did not suffer from the problem. This object added a new interface to the kernel, allowing user programs to create a partitioned memory region if they specified extra parameters.

Adding a new piece of code to the kernel and making it available through a new interface is the simplest case of dynamic update, because there is no need to replace old code or transfer state information. This update was implemented as a simple loadable module, consisting of the code for the new region object and some initialisation code to load it and make it available to user programs. This module could be shipped with programs requiring it, or could be loaded into the kernel on demand when a program required the new interface, with either option avoiding a reboot.

This scenario demonstrates the use of a module loader combined with an extensible kernel interface to add new functionality to a running kernel. There is nothing inherently new here; existing systems also allow modules to be loaded, for example to provide new file systems or device drivers. However, K42's modularity makes it possible to replace a portion of the page-fault path for a critical application with a new set of requirements without affecting the path taken by other applications, which could not be done on an existing system such as Linux.

4.3.2 Fix for memory-allocator race condition

This scenario involves a bug fix to a kernel service, one of the key motivations for dynamic update. In the course of development, a race condition had been discovered in the core kernel memory allocator (in the method PageAllocatorKernPinned::getNextForMDH()) that could result in a system crash when kernel memory was allocated concurrently on multiple CPUs.

Fixing this bug required adding a lock to guard the allocation of memory descriptors, a relatively simple code change. In fact, only three lines of code were added, one to declare the lock data structure, one to initialise it, and another to acquire it (and automatically release it on return). A recompile and reboot would ordinarily have been required to bring the fix into use. However, even while memory allocation and deallocation was occurring, the dynamic update was applied without the need for a reboot using the new functionality.

The replacement code was developed as a new class inheriting almost all of its implementation from the old buggy object, except for the declaration of the lock and a change

```
class PageAllocatorKernPinned_Update : public PageAllocatorKernPinned {
public:
    BLock nextMemDescLock;                                  // new lock structure

    void pinnedInit (VPNum numaNode) {
        nextMemDescLock.init();                             // initialise lock
        PageAllocatorKernPinned:: pinnedInit (numaNode);    // call original method
    }

    virtual void* getNextForMDH(uval size) {
        AutoLock<BLock> lock(&nextMemDescLock);             // unlock on return
        return PageAllocatorKernPinned::getNextForMDH(size); // call original method
    }

  DEFINE_GLOBALPADDED_NEW(PageAllocatorKernPinned_Update); // K42ism
};
```

Figure 4.11: Update source code for memory allocator race condition.

to the function that acquired and released it. This caused the C++ compiler to include references to all the unchanged parts of the old class in the replacement object code. Simple implementations of the state transfer functions were also provided to allow the object to be hot-swapped. The key parts of the replacement code are shown in Figure 4.11.

The new class was compiled into a loadable module, and combined with initialisation code that instantiated the new object and initiated a hot-swap operation to replace the old, buggy instance. Because this object was a special-case object with only a single instance, it was not necessary to use the factory mechanism.

This scenario demonstrates the use of hot-swapping as part of the dynamic update mechanism, combined with a kernel module loader, to dynamically update live code in the system.

4.3.3 File cache manager optimisation

This scenario involves an optimisation to the implementation of file cache manager objects. An FCM is instantiated in the K42 kernel for each open file or memory object in the system.

It had been discovered that the unmapPage() method did not check if the page in question was already unmapped before performing expensive synchronisation and IPC operations. These were unnecessary in some cases.

```
class FCMDefault_Update : public FCMDefault {
public:
    /* define a factory for the updated class */
    DEFINE_FACTORY_BEGIN_VERSION(Factory, 1)
        /* standard create method, matches FCMDefault::Factory::create() */
        virtual SysStatus create(FCMRef &ref, FRRef frRefArg, uval pageable,
                                 uval preFetchPages = 0,
                                 uval maxPages = FCM_MAX_NUMPAGES);
        /* method used by the update code to create a replacement instance of the
           class for updating old instances */
        virtual SysStatus createReplacement(CORef ref, CObjRoot *&root);
    DEFINE_FACTORY_END(Factory)
public:
    /* method changed from previous version */
    virtual SysStatus unmapPage(PageDesc* pg) {
        if (pg->mapped) { // only call unmapPage() if the page is mapped
            return FCMDefault::unmapPage(pg);
        } else {
            return 0;
        }
    }
    DEFINE_GLOBALPADDED_NEW(FCMDefault_Update);
};
```

Figure 4.12: Update source code for file cache manager optimisation.

```
static void do_update(void) {
    /* locate old factory */
    FactoryRef factory = FCMDefault::Factory::getFactoryRef();
    /* use static method on updated factory class to initiate update of factory */
    SysStatus rc = FCMDefault_Update::Factory::Update((CORef)factory);
    /* check for success */
    tassertMsg(_SUCCESS(rc), "Factory::Update failed \n");
}

extern "C" {
    /* kernel module's initialisation function */
    void _init(void) {
      do_update();
    }
}
```

Figure 4.13: Update module initialisation code for file cache manager optimisation.

A new version of the FCM object that performed the check before unmapping was developed, and prepared as a loadable module. Applying this update required the factory mechanism, because the running system had FCM instances present that needed to be updated, and because new instantiations of the FCM needed to use the code in the updated module. The source code for the updated object is shown in Figure 4.12; it includes a new factory for the updated class with a version number of 1. The module initialisation code for this update is shown in Figure 4.13; it triggers the dynamic update process by locating the old factory instance and invoking a static update method on the new factory.

4.3.4 File-sync patch

This example illustrates a more complex update involving significant changes to many data structures, and thus requiring the use of factories and state transfer functions. When a file is closed, or when its last mapping is removed, the kernel initiates a *sync* operation to flush modified blocks to disk. This is known as an *unforced sync*, as opposed to the *forced sync* 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.

This patch was converted 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, the key parts of which are shown in Figure 4.14.

This update also demonstrates the use of the factory mechanism, along with state transfer functions to achieve significant changes in an object's data structures. As will be detailed in Chapter 6, this update was also used to test lazy update in a loaded system.

4.3.5 Use of adaptors in page allocator change

This example shows a scenario where adaptor objects are required to support changes to interfaces in the virtual memory system, and where multiple objects are affected by a change.

To improve the performance of K42 in the case of memory pressure, and to fix an unhandled condition that could cause the kernel to run out of memory and panic, a number of changes were made to the core kernel memory allocator class (PageAllocatorKernPinned). These changes included the addition of a new method called by the *root page manager* (PMRoot) class, which was also modified as part of the change.

```
SysStatus FCMFile_Update_Root::dataTransferImport(DataTransferObject *dtobj, DTType
    dtt, VPSet dtVPSet) {
    tassertMsg(dtt == DTT_FCM_FILE, "unhandled import type\n");
    if (dtt != DTT_FCM_FILE) return -1;
    FCMFile *oldfcm = (FCMFile *)((DataTransferObjectFCMDefault *)dtobj)->fcm();
    FCMFile_Update *newfcm = (FCMFile_Update *)therep;

    /* transfer common data by copying it */
    newfcm->frRef = oldfcm->frRef;
    newfcm->pageSize = oldfcm->pageSize;
    newfcm->referenceCount = oldfcm->referenceCount;
    newfcm->pmRef = oldfcm->pmRef;
    newfcm->pageable = oldfcm->pageable ? 1 : 0;
    newfcm->backedBySwap = oldfcm->backedBySwap ? 1 : 0;
    /* ... lines cut for brevity ... */
    newfcm->giveBackNumPages = oldfcm->giveBackNumPages;
    newfcm->regionList = oldfcm->regionList;
    newfcm->pageList = oldfcm->pageList;
    newfcm->pageList.fixUpCopyHack();
    newfcm->segmentHATList = oldfcm->segmentHATList
    newfcm->rrq = oldfcm->fsyncRQ;

    /* see if there are outstanding sync requests in progress */
    if (!oldfcm->fsyncQueue.isEmpty()) {
        struct FCMFile::fsyncQueueItem *fqi;
        void *ptr = NULL;

        /* add any blocked threads to our syncThreads list */
        while ((ptr = oldfcm->fsyncQueue.next(ptr, fqi))) {
            if (fqi->type == FCMFile::fsyncQueueItem::FSYNC_Q_THREAD) {
                newfcm->syncThreads.add(fqi->data.thread);
            }
        }

        /* prepare to restart sync operations */
        if (newfcm->syncThreads.isEmpty()) {
            newfcm->restartSyncOps = SYNC_OPS_NON_FORCED;
        } else {
            newfcm->restartSyncOps = SYNC_OPS_FORCED;
        }
    }

    delete dtobj;
    return 0;
}
```

Figure 4.14: State transfer code for file-sync patch.

```
class MyAdaptor : public CObjRepArbiterAdaptor {
protected:
    virtual SysStatus handleCall(CORef targetRef, CallDescriptor* cd, uval fnum) {
        if (fnum >= 54) fnum++; /* new method at position 54 in vtable */
        return CObjRepArbiterAdaptor::handleCall( targetRef , cd, fnum);
    }
    DEFINE_LOCALSTRICT_NEW(MyAdaptor);
};
```

Figure 4.15: Adaptor code for a new method in a class.

Adding a method to an existing class requires the use of an adaptor, because the virtual function table layout and virtual method numbers used for that class are changed. The adaptor code for this change is shown in Figure 4.15. For any call on the old interface with a method number greater than or equal to the method number of the added method, the method number is incremented before passing the call on to the new object. The position of the added method (54, in this case) was manually determined by consulting the compiled object code, however this could be automated.

As part of the overall update for such a change, once the PageAllocatorKernPinned object has been updated and the adaptor installed, it is then possible to update the PMRoot object to a version that depends on the change and uses the new interface. This example shows how, in the model developed for dynamic update, one logical change is implemented as a series of updates, allowing the system to continue making progress.

4.4 Conclusions

This chapter has presented the design and implementation of dynamic update for the K42 operating system. The implementation builds upon K42's hot-swapping feature to achieve a whole-system dynamic update as a series of self-contained hot-swap operations. It satisfies the requirements developed in the previous chapter, by using factory objects to perform state tracking and coordinate the dynamic update process. Updates affecting many data structures are performed lazily, and changes to object interfaces are supported through the use of adaptors. Finally, to show how a dynamic update is constructed and applied within the system, several updates of increasing complexity that were used to test the system were described, and the relevant source code presented.

The following chapters build upon the work presented here; Chapter 5 describes an analysis of the K42 CVS revision history to validate the applicability of the update system developed here, and Chapter 6 reports the results of performance measurements of the implementation.

5 CVS History Analysis

This chapter describes a study performed on changes from K42's CVS revision history to analyse the applicability of the dynamic update system whose design and implementation was covered in the previous chapter. Questions the study seeks to answer include:

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

In answering these specific questions, the study also aims to determine more generally the possible properties of changes to OS code, and to develop a framework for categorising and analysing these changes.

Section 5.1 gives an overview of the method used for the analysis. Following this, Section 5.2 describes the results of a broad automated analysis of every change in the revision history, before Section 5.3 covers a more detailed manual analysis of a subset of the revision history. Section 5.4 concludes.

5.1 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 relevant core modules.[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, mechanisms and heuristics first had to be developed for extracting independent transactions or changes from CVS revisions. This required two assumptions:

[1] The kitch-core and kitch-linux CVS modules were examined.

- It was assumed that each commit operation by a developer contained a single logical change or feature. This is usually true, but not always. A few developers tended to commit unrelated changes together. This leads to fewer and larger changes being analysed, so the results are more pessimistic than they might otherwise be.
- It was assumed that after each commit the repository was in a consistent state. That is, it could be expected to compile and run correctly. Because developers make mistakes, 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, a tool was developed to process the CVS repository data and import it into a database for further analysis. The steps taken by this tool were as follows:

1. The first step was to parse all the data from CVS into a database. For this, a modified version of the *slurp* tool [MLR+04] was used. This tool crawls across the repository, reading the contents of every RCS file and storing the meta-data and change text of every revision. Any changes on a branch were ignored, because branches are not significant in K42's development history, and including branches would complicate the later phases of the analysis. The commit message information was also stored in a separate table, and MD5 hashes were used to detect duplicate messages. This assists with the next step.
2. The next task was to group related revisions into logical transactions. The following heuristic was used:

   - Sort all the revisions by author, time-stamp and log message, and process them in that order.
   - As long as the author and log message are the same, and the time-stamp remains within a sliding window of 200 seconds, group the revisions into a transaction. This means that a transaction consists of revisions that all have the same author and log message, and that every revision in the transaction was made within 200 seconds of at least one other revision in the transaction.

   This algorithm is as described by Zimmermann and Weißgerber [ZW04], and the window of 200 seconds is the same. It also gave accurate results for K42, although because every revision had a good log message, the use of a time window was not strictly required.[2]

[2] Despite all commits having good log messages, the use of a time window still made minor differences to the grouping of transactions. It enforced separation of regular automated commits from a nightly regression test system, and also separated a series of related commits from one developer that were made hours apart, but with the same log message.

3. For each revision of each file, the full text of that revision was then generated from the RCS information stored in step 1. For each source file revision, all the comments were also filtered out, the code reformatted in a consistent style using an *indent* tool, and the differences computed between the cleaned and reformatted source. This significantly reduced the number of irrelevant changes that needed to be examined.

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

For both analyses only transactions that altered some kernel code were considered. 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.

5.2 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.

Determining what functions, data structures, and objects were changed by each transaction in the repository would imply 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, a pseudo-parser that handles just enough of the language (for example, the class keyword, function definitions, and braces) was written to identify a *program context* for every line of a C++ source file. A program context is the name of a function or class, or a special global context (used, for example, for preprocessor directives).

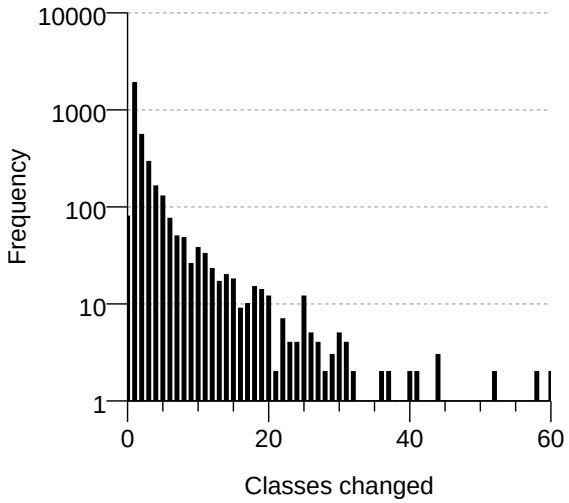

Figure 5.1: Number of classes modified in a CVS transaction. The large majority of changes alter fewer than ten classes, but changes to multiple classes are significant.

Given this contextual information, and the differences computed earlier, any classes or functions added or deleted by each transaction, and also any that were modified, were identified. Every transaction that included a change to kernel code (a total of 3,618 transactions) was then examined and categorised based on those that added contexts, those that deleted them, and those that just modified code within existing contexts.

The class name of each context that was changed was also determined. The average transaction changed two classes, but this is skewed by a number of high outliers; the median transaction changed only one class, and the large majority of transactions affect fewer than 10 classes, as shown in Figure 5.1.

Using a list of classes known to be dynamically updatable, the transactions that changed only dynamically-updatable code were then identified. The list of dynamically-updatable classes included classes which do not yet have state-transfer functions or factories, so are currently not updatable. These were included, because the addition of state-transfer functions and factories is a relatively simple change (its effects are confined only to the class itself), and because it was believed that showing the limitations of the model and system is more meaningful than showing those of the K42 implementation.

It was found that 22% of transactions only modified or added methods in dynamically-updatable classes. Upon closer examination, a number of common problems were found to be preventing 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 the 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 variable declarations, that would be handled in other ways for a dynamic update.

If these are included, the result rises to 48% of transactions that are dynamically updatable. An additional 3% delete classes or methods, and are probably updatable using adaptors, but without looking at the specific cases it is impossible to be sure. If changes before 2002 are excluded, 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 all the commits in the K42 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 a reasonable lower bound for the proportion of dynamically-updatable changes. For a more accurate result, the manual analysis described in the following section was conducted.

5.3 Detailed study of sample set

The automatic analysis gave 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 the dynamic update system to the specific types of changes which are most relevant to the problem, a manual investigation of a sample of the CVS transactions from K42 was conducted.

A simple web application was developed, allowing a human analyst to examine randomly-selected transactions from the database. 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 base, as well as an understanding of what could be changed by a dynamic update. The goal was to examine a sufficiently large number of transactions to obtain conclusions about the proportion of dynamically-updatable changes. In total, 250 transactions were manually analysed.

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 examined transactions were ignored.

The change as a whole was then examined, and placed 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 source code. Of the non-ignored transactions, 48% were restructuring, 36% added new features, 11% were bug fixes, and the remaining 5% performance improvements. No security fixes were found. 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 [ABB+05], one can expect that the results for the bug fix category will be a good indicator of the system's support for security updates.

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

Finally, the analyst decided whether the change was convertible to a dynamic update. For each transaction, there were 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, or the change could not easily be updated. The results are shown in Table 5.1 and summarised in Figure 5.2.

Of the transactions categorised as bug fixes or performance improvements, 79% could be converted to dynamic updates, with slightly more than half requiring the use of an adaptor (none required rewriting). All of the remaining 21% of maintenance changes that could not be updated were changes to low-level code in the exception handlers, that is not part of the normal object system and thus not dynamically updatable.

change type	number	simple update	adaptor update	rewrite update
maintenance	24	12 (50%)	7 (29%)	0
new features	54	6 (11%)	7 (13%)	6 (11%)
restructuring	72	4 (6%)	6 (8%)	1 (1%)

Table 5.1: Results of manual CVS analysis.

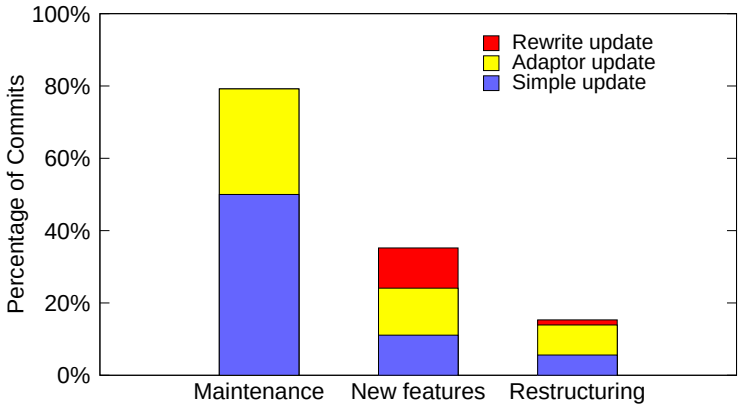

Figure 5.2: Results of manual CVS analysis.

Of the other categories, 35% of new features and 15% of code restructures could be converted to dynamic updates. Combining all the categories, only 33% of all changes could be converted to dynamic updates, but this is skewed by the large proportion of new features and restructuring changes seen in K42's revision history, and as has been previously discussed, the focus of this work is on maintenance patches. 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 the analysis, it was found to be quite 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 dynamically updatable, or was part of another object. These other changes would have been ignored in developing the dynamic update, so the same was done in the analysis, and another series of flags were added to note when this occurred. Of all the dynamically-updatable changes, 39% fell into one of these categories. Of only the bug fixes and performance improvements, minor parts of 33% of the changes were ignored.

5.4 Conclusions

This chapter has presented an analysis of the K42 revision history, including an automated analysis of every change in the CVS repository, as well as a detailed manual analysis of a sample of the changes. Although relatively few of the manually-analysed changes were maintenance changes, 79% of them could directly be converted to dynamic updates.

Beyond the specific results related to K42, the work presented in this chapter has developed a framework for analysis and categorisation of changes to OS code. It has also built upon previous work to develop tools and techniques for performing that analysis in a semi-automated fashion.

Chapter 8 will discuss these results further, including a list of and solutions to some of the common problems that prevented changes from being classified as dynamically updatable, and strategies for updating the non-modular code that accounts for the remaining 21% of maintenance changes. In the following chapter, a series of performance measurements are presented that analyse the overheads and costs incurred by the dynamic update system.

6 Performance Measurements

In this chapter, the results of experiments conducted to measure the overhead and performance of the update system are reported. Section 6.1 examines the extra overhead required for dynamic update functionality, Section 6.2 reports on the time taken to apply various updates, and Section 6.3 concludes.

For macro-benchmarking, the ReAIM implementation of the AIM7 multi-user benchmark was used 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: the test using Unix-domain sockets was replaced with UDP sockets, some code was altered to handle different error return values from K42's Linux emulation library, and the benchmark was prevented from removing shared memory regions at the end of its run, because this is not yet supported by K42. The benchmark was run inside a RAM disk, to avoid any 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. K42 was built in the no-debug configuration, and ran in dual-processor mode.

6.1 Dynamic update overheads

In this section the added runtime costs of having support for dynamic update in the system are examined. This is much more important than the time to apply an update (which is measured in Section 6.2) because updates are expected to be infrequent events, and because even if the system experiences a slowdown while an update is applied, the advantages over rebooting are significant.

6.1.1 Indirection overhead

First, a property that is part of the fundamental structure of K42 is considered: 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 [Gam99]. The added instruction is a dependent

object	static create	factory create	overhead
dummy	1.3μs	1.4μs	7.7%
FCM	2.0μs	2.1μs	2.8%
process	13.4μs	13.5μs	0.70%

Table 6.1: Cost of creating various kernel objects with and without a factory.

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, the overhead of object indirection can be estimated 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, the overhead of indirection is calculated to be 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 prevents the use of branch prediction features in modern hardware. 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.

6.1.2 State-tracking overhead

A factory is now involved during object creation, to record the new object's reference. During deletion, the reference is removed from the factory's data structures. Presently, the factory implementation uses a simple list. Although this has higher deletion costs than other data structures, it has yet to pose a problem. If the deletion overhead becomes a bottleneck, the list could be replaced with a more suitable data structure. The factory also results in slightly higher memory utilisation, of an additional two words per object.

Micro benchmarks

This section reports measurements of the cost of creating three different objects using either a factory or a statically-bound method. This includes allocating storage space for the object, instantiating it, and invoking any initialisation methods. Each test was repeated

100,000 times, and the total time measured using the processor's cycle counter. The results are summarised in Table 6.1.

The first object, a dummy object, was created specifically for this test, and encapsulates a single integer value. This result represents the worst-case for adding a factory, with an overhead of 7.7% over the base creation cost of 1.3μs. The next object is a file cache manager, an instance of which is maintained by K42 for every open file in the system. Creating an FCM object is only 2.8% slower with a factory, but this number overstates the true impact, because in practice FCM creation is usually followed by a page fault, which must be satisfied either by zero-filling a page, or by loading it from disk. Finally, the cost of creating a process object was measured, a relatively expensive operation because it involves initialising many other data structures. The results show that in such a case the additional cost imposed by the factory was a low 0.7%, even before considering the additional costs involved in process creation such as context switches and initial page faults. For all objects, the factory overhead was approximately constant, at around 100ns or 200 cycles.

System throughput with ReAIM benchmark

To measure the impact of factories on system performance, the ReAIM benchmark described previously was used. Factories were implemented for process objects, one of which exists for each process, and the FCM and FR objects, 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, any impact would be expected to be visible.

The unmodified system and the modified system with factories added for the above objects were both tested with the ReAIM benchmark. The results of this experiment are shown in Figure 6.1. Comparing the average throughput of each run in jobs per minute, they show a very slight (less than 0.5%) performance improvement with the factories present. This is within the noise caused by changes in code layout and cache behaviour, leading to the conclusion that the use of factories within the kernel has negligible performance impact on the overall system. The same conclusion was also reached on different hardware using the SPEC SDET multi-user system throughput benchmark [BHA+05].

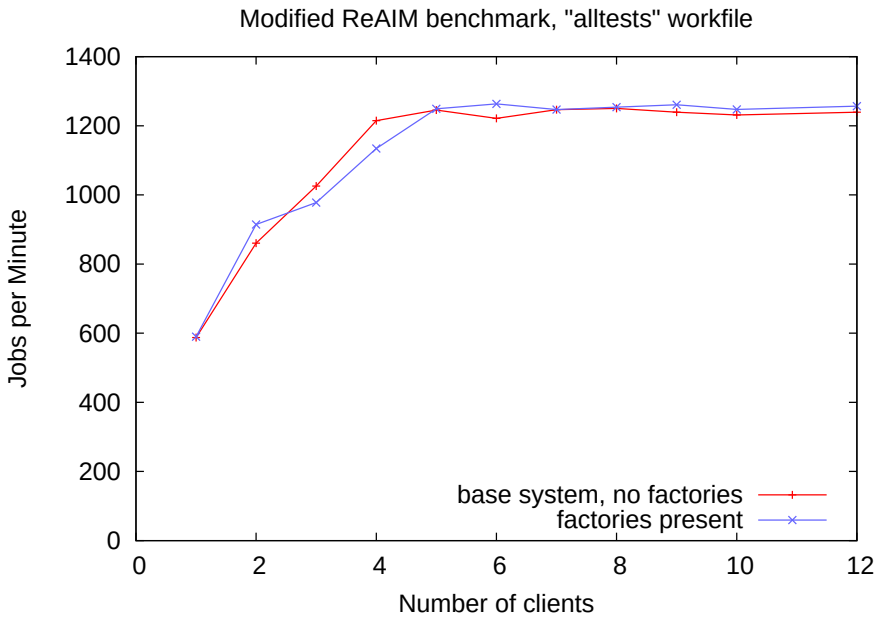

Figure 6.1: Results of ReAIM benchmark with and without factories. A slight performance improvement is seen with the factories present.

6.2 Time to apply updates

6.2.1 Time to apply file-sync update

Figure 6.2 shows the result of applying the file-sync update described in Section 4.3.4 during a ReAIM benchmarking run. ReAIM incrementally tests higher numbers of clients, so wall-clock time in this graph runs from left to right. The update was initiated 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 in a separate run of the benchmark, a hardware cycle counter was used 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 object and convert the existing factory, and then 487μs to mark all the object instances to be lazily updated. At no point was the whole system's execution blocked.

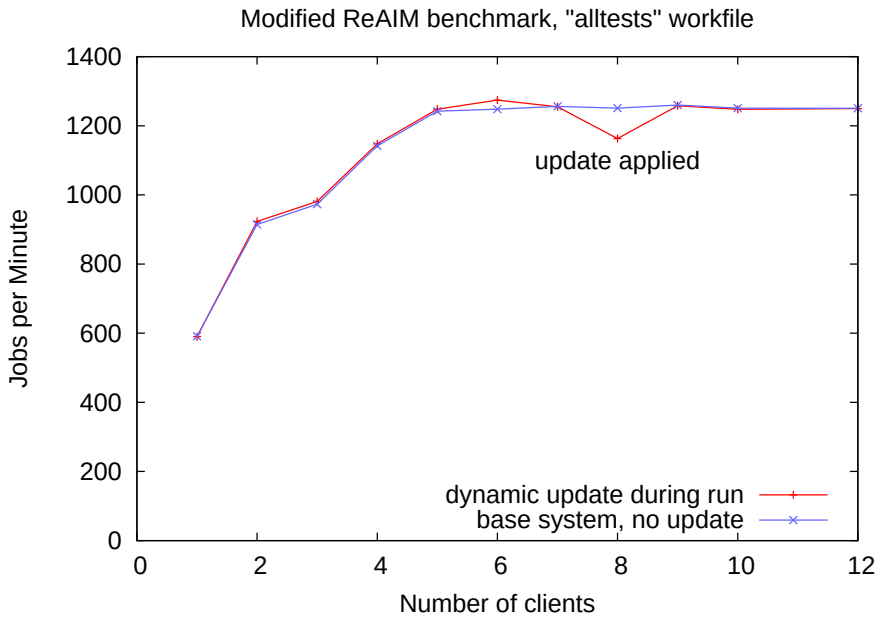

Figure 6.2: Results of applying the file-sync update during a ReAIM benchmark run. The update was applied during ReAIM's test of eight clients.

This update was also used to time the 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.

6.2.2 Cost of adaptor objects

The cost of using an adaptor object was measured with a micro benchmark. A null adaptor was used, which simply passed the call on unmodified to the underlying object. The time taken for a direct object call was 10ns, and for a call through the null adaptor 124ns, an overhead of 114ns or 228 cycles. This extra time is required for changing stacks to invoke the adaptor code and then forwarding the call to the underlying object, mechanisms provided by the arbiter framework on which adaptors are implemented [Fin06]. A dedicated and optimised adaptor implementation would likely reduce the overheads.

Most adaptors are relatively simple, and add only a few instructions to the base cost, to load default arguments or change method numbers. For example, an adaptor that shuffled method numbers showed no measurable difference in performance from the null adaptor.

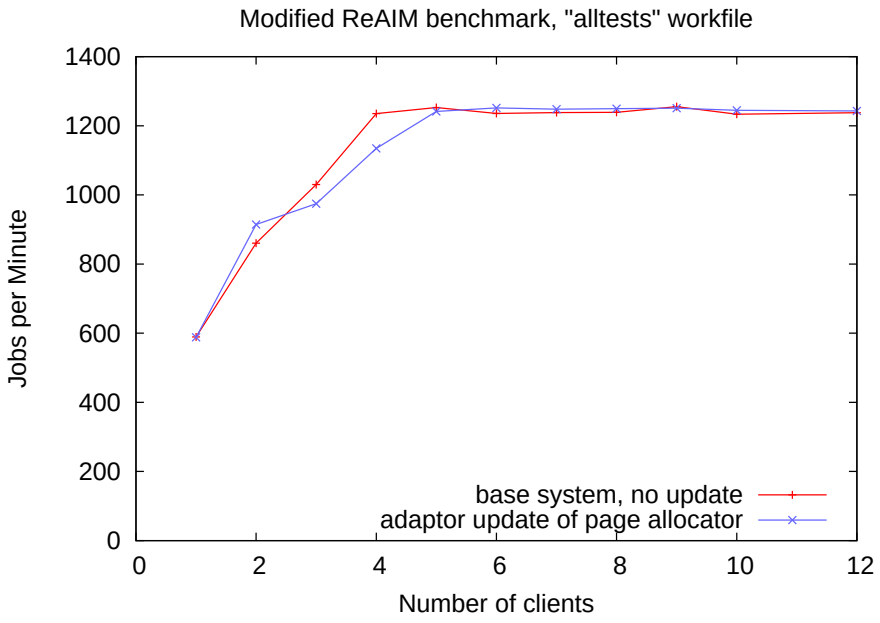

Figure 6.3: Results of applying an adaptor update to the kernel page allocator during a ReAIM benchmark run. The update was applied during ReAIM's test of eight clients.

The overall costs of using an adaptor in a dynamic update are dependent on the complexity of the update, in terms of the number of objects updated with an adaptor, and the frequency of calls to those objects. Figure 6.3 shows the result of applying the update to the kernel page allocator described in Section 4.3.5 during a ReAIM benchmark run. As in the previous experiment, the update was initiated when there were eight clients active, however in this case the overhead imposed by the single adaptor does not affect the overall system throughput.

To show results closer to a worst-case scenario for adaptors, an experiment was constructed that installed a dummy adaptor on each FCMFile object during a ReAIM benchmark run. The results of this experiment are shown in Figure 6.4, where a small overhead can be seen for the use of the adaptor. In a real update scenario, this overhead would disappear as the relevant client objects were updated and the adaptors removed.

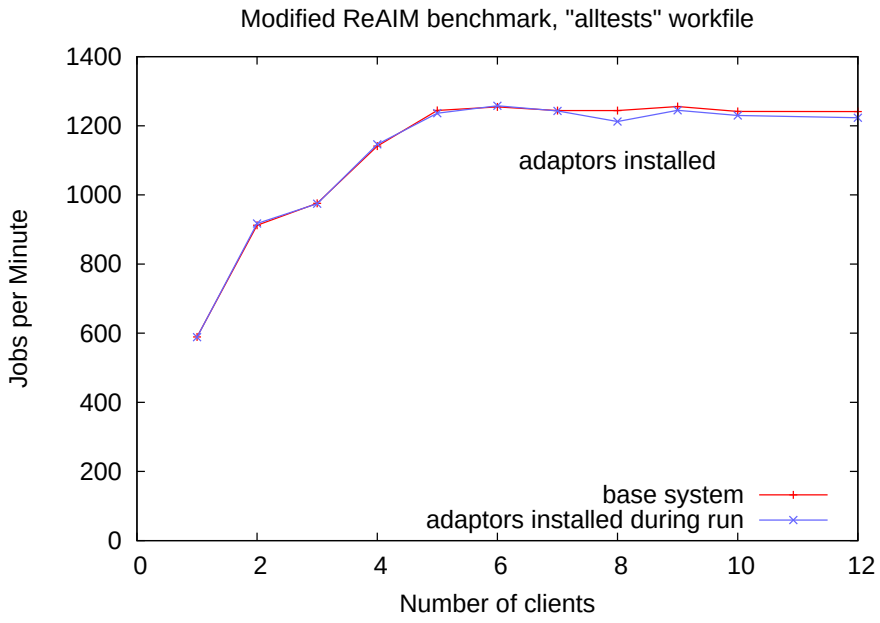

Figure 6.4: Results of installing a dummy adaptor on each FCMFile object during a ReAIM benchmark run.The adaptors were installed during ReAIM's test of eight clients.

6.3 Conclusions

The results reported in this chapter show that the extra levels of indirection required for dynamic update, in the form of the object translation table and factory objects, impose negligible performance impact on system throughput. K42 suffers less than 1% runtime overhead when updates are not being applied. Furthermore, when updates are being applied, the performance impact is moderate. The ReAIM benchmark experiences a slowdown in the vicinity of 10% while applying the complex FCMFile update, and returns quickly to its previous throughput as the update completes. Finally, adaptor objects do not impose measurable overhead for small updates, and in updates to many objects, their costs are acceptably low.

The following chapter will extend the approach taken in K42 to a more general model for dynamic update to commodity operating systems such as Linux and FreeBSD.

7 Dynamic Update in Other Systems

The previous chapters have focused on the case-study design, implementation and evaluation of dynamic update for the K42 experimental operating system. However, one of the goals of this work was to develop a model for dynamic update to existing commodity operating systems. This chapter discusses how the design for dynamic update could be generalised and applied to other modular operating systems, with a focus on two common open-source UNIX-derived systems: Linux [BC02] and FreeBSD [MBK+96, Fre06].

First, Section 7.1 outlines the basic approach, which is based on updates at the granularity of kernel modules. Next, Section 7.2 revisits the requirements developed previously and shows how these could be satisfied. Finally, Section 7.3 discusses the limitations of applying the model to commodity systems, and Section 7.4 concludes.

7.1 Approach

As already discussed, the model adopted for this work was designed to achieve a dynamic update of the operating system as a series of self-contained updates to modular kernel code. Modern operating systems such as Linux and FreeBSD are being constructed in an increasingly modular fashion to allow for features such as improved customisability, as was discussed in Section 2.2.1, suggesting that the modular approach for dynamic update may apply to such systems.

Both Linux [dGdS99] and FreeBSD [Rei00, Fre06, chapter 9] support loadable kernel modules. At a high level, the process of loading a kernel module in both systems is similar, and proceeds as follows:

1. A kernel module is a loaded into a free area of kernel memory, and the kernel's dynamic linker is used to link unresolved references in the module to existing symbols in the kernel.
2. The module provides an initialisation function, that is called by the kernel once loading and linking is complete.
3. The initialisation function performs setup and registration tasks, which depend on the specific functionality of the module.

This step requires a form of extensibility to be implemented in the relevant kernel subsystems. For example, the PCI subsystem allows device drivers to register themselves by providing the device identifiers that they support, and function pointers to the code within the kernel module that implements the driver. Other subsystems use different but similar interfaces, generally consisting of tables of function pointers with a standard interface.

Code within loadable kernel modules is invoked indirectly via function pointers that are registered and called in a manner determined by the API of the relevant kernel subsystem. Unlike K42's object translation table, there is no standard abstraction for invoking code in a module; once it has a function pointer to a module's code, another module may call that function directly.

However, although they do not provide the same consistent mechanisms as K42's object system, modular parts of Linux and FreeBSD such as the VFS layer and device-driver interfaces are effectively object-oriented, providing data hiding and indirection. For example, file systems are invoked through a table of function pointers held in the vnode structures, and device drivers use similar interfaces. This provides the same indirection as K42's object translation table, although at a coarser granularity. These structures of function pointers can be used in the same way as K42's object translation table; by overwriting the function pointers, calls to the module can be interposed or redirected.

Using this observation as the basis of the approach for applying dynamic update to a modular OS, the following sections discuss the design and implications of such an approach.

7.2 Requirements

In Section 3.1, the fundamental requirements for a system to support module-based dynamic update were identified. These requirements are now revisited, with a discussion of how they can be provided in a commodity operating system, using the examples of Linux and FreeBSD.

Modularity

Modularity is already widely used in constructing operating systems, with the virtual file system layer [Kle86] and streams facility [Rit84] being well-known examples. In Linux, loadable kernel modules are now supported for device drivers, file systems, some networking functionality, and security policies. FreeBSD allows kernel modules to be used for the same subsystems as Linux, as well as supporting a fully-modular networking subsystem, and modular disk IO layer [Lav05]. This support could be leveraged to provide dynamic update capabilities in the same areas where kernel modules can be used.

Safe point

In order to update a module, it must generally be ensured that there are no kernel threads that are executing within any of the the module's functions. In the current Linux and FreeBSD implementations of kernel modules, each module has a reference count, incremented when the module is used by another part of the kernel (for example, when a file system is mounted, the relevant module's usage count is incremented). However, there is no indication of when the module's code is being executed—it is possible and common for a module to have a non-zero reference count, but not have any threads executing within the module. If dynamic updates were only allowed when the reference count was zero, then the dynamic update system would be no better than what can be achieved already by unloading and reloading a kernel module.

Using the tables of function pointers through which module code is called, mediator code can be interposed as it is in K42; however, K42's thread-based quiescence detection mechanism is feasible in neither Linux nor FreeBSD, and RCU in Linux [MSA+02, MS98] operates only around critical sections such as locks, so an alternative mechanism to achieve safe points for updates is required. Two possibilities include checking the stack of all running kernel threads to determine if any are executing inside the module, or adding atomic counters at function entry and exit points.

Checking the stack of kernel threads would involve walking through the stack frames of every kernel thread in the system to determine which, if any, were executing within the affected module. The problem with this approach is that, although it does not affect the system's base-case performance, it would significantly delay the critical phase of an update in which new calls are blocked. It would also scale poorly as more kernel threads are added.

Adding a module use count, that is incremented and decremented on entry to and exit from all externally-callable functions in a module, seems the best option. This will add a small cost to module invocations, but in return will enable the efficient detection of quiescence by waiting for the counter to reach zero.

State tracking

State tracking is required when data structures with multiple instances, such as a file system's inode structures, are to be changed during a dynamic update. To track such state, a module would be required to incorporate a factory; for example, by adding each newly-allocated structure to a list, if it doesn't already do so. In practice, most modules already track state information internally, so adding factory support merely requires implementing a common interface to access that state.

Other work on dynamic update for Linux has relied on purely ad-hoc mechanisms for state tracking [CCZ+06, MR07].

State transfer

Unlike K42's generic state-transfer mechanism, which supports both dynamic update and hot swapping, state transfer in a dedicated dynamic update system can be much simpler. Because dynamic update almost always involves minor changes to data structures, and because the new version of a module can be expected to understand the data structures of the previous version, state transfer in this case can be performed using the internal data structures of the previous version, relying on the updated version to perform any changes necessary. This would require data import functions to be implemented in updated modules.

One alternative to explicit state transfer code is suggested by Nooks [SBL03, SAB+04], which supports recoverable device drivers on Linux. In this system, special *shadow drivers* monitor all calls made into and out of a device driver, and reconstruct a driver's state after a crash using the driver's public API. Only one shadow driver must be implemented for each device class (such as network, block, or sound devices), rather than for each driver. Because many device drivers exist for each device class, the implementation work needed to support state transfer is lower than that involved in writing state transfer functions for each device driver. The main drawback of this approach is that there is a runtime overhead imposed by the use of shadows, unlike conversion functions, which are only invoked at update time. Furthermore, although this approach works for device drivers, where few shadows must be implemented relative to the number of drivers, in other parts of the kernel the same scaling effect does not apply.

Redirection of invocations

Although few commodity systems include a uniform indirection layer equivalent to K42's object translation table, most systems including Linux and FreeBSD use function-pointer indirection wherever kernel modules are supported. These function pointers could be used to support dynamic update, however, the lack of a uniform indirection mechanism would limit the applicability of dynamic update to those specific areas of the system for which it was implemented.

FreeBSD includes a kernel objects (Kobj) interface that provides an object-oriented programming model for C code within the kernel [Fre06, chapter 3]. It defines standard functions and macros for creating objects and invoking methods on them. However, the Kobj interface is currently only used for some device driver classes, namely serial drivers

and sound drivers, disk geometry modules, and the kernel linker. If the use of this interface within FreeBSD becomes more widespread, it would offer an excellent place to implement the interposition, indirection, and factory mechanisms required for dynamic update.

For dynamic update to multiple-instance data structures to be supported, it is desirable that each instance be individually updatable. For example, the VFS layer's use of one set of function pointers per vnode, rather than one set for the entire file system, allows a file system's data structures to be converted incrementally. The alternative would be to block all access to all file system nodes while they are updated, a potentially much longer-running operation.

Version management

Existing systems already maintain version information for kernel modules. For example, each Linux module carries attributes indicating the version of the kernel it was compiled for, and a list of any other modules on which it depends. FreeBSD modules also include a version number for the module itself, and versions of any modules on which it depends. These attributes could be reused for dynamic update version management.

7.3 Limitations

The biggest limitation of this approach is its dependence on modularity and the use of indirection to update module code. Although parts of most commodity systems, such as the device driver and file system APIs, are modularised, often core parts of the kernel are not. In particular, neither Linux nor FreeBSD currently support loadable modules for the virtual memory system or the scheduler, although FreeBSD's kernel is in general more clearly modularised than Linux, and thus would be easier to convert to supporting loadable modules and dynamic update features. The dynamic update system proposed by this work is dependent on modularity, so cannot update code that is not modularised.

One significant difference between K42 and most UNIX-like systems such as Linux and FreeBSD is blocking threads. K42 kernel threads are short-lived and non-blocking, allowing the mediator to block access to an object and simply wait until existing threads terminate to achieve quiescence. In Linux and FreeBSD, however, system call handlers may block for IO or other long-running operations. One possible but inelegant solution is, when applicable, to abort system calls with EINTR (interrupted call) or EAGAIN (resource temporarily unavailable) errors, from which an application should recover by retrying the call. If a blocking call cannot be interrupted, the update must be delayed and retried until it completes. This could be avoided by a wrapper to convert uninterruptible blocking system calls into restartable variants such as select().

7.4 Conclusions

The main limitation in applying this approach to Linux, FreeBSD, and similar systems is their current extent of modularity. For example, unlike K42, core parts of Linux such as the scheduler and virtual memory system are not modular. Despite this, a dynamic update implementation for a commodity system such as FreeBSD appears feasible. In particular, it is not necessary to apply modularisation and dynamic update infrastructure pervasively throughout the kernel. Rather, dynamic update can be enabled incrementally for specific subsystems, by adding interposition and state tracking (or subverting existing structures such as FreeBSD's Kobj interface). Along with other projects [SAB+04, WRA05], dynamic update provides further motivation for increasing the modularity of commodity kernels.

A small number of other systems have attempted to achieve dynamic update in Linux. In the following chapter, Section 8.2 will compare the approach outlined here to that taken by the related work, and discuss the implications of the different design decisions.

8 Discussion

This chapter discusses the findings and implications of this work with reference to the research design of Chapter 3, and also attempts to resolve some of the outstanding issues.

The hypothesis of this dissertation, as stated in Chapter 3, was that dynamic update for operating systems could be achieved through the use of modularity, that by building dynamic update mechanisms around the existing modules within an operating system, and performing a dynamic update as a series of individual modular updates, it would be possible to dynamically update the operating system as a whole. The implementation of the K42 dynamic update system, as detailed in Section 4.2, and the successful application of the test updates of Section 4.3 have confirmed this hypothesis as it relates to K42 and those sample updates.

One desired property of the dynamic update model was that it should support all kinds of changes commonly required for maintenance, including bug fixes and security patches. To assess this, Chapter 5 presented the results of a study of K42's CVS repository that analysed the applicability of the dynamic update mechanism to the changes in K42's revision history. Extending those results from a case-study analysis of the revision history to more general conclusions about the applicability of the dynamic update model is potentially error-prone, because 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, the study gives an indication of the updates seen in systems code. In a production operating system in maintenance mode, one would expect far fewer broad restructures and added features, and a greater proportion of performance and security updates or bug fixes. The results in the manual analysis showed that approximately 79% of maintenance changes could directly be converted to dynamic updates. This can be regarded as a worst-case for the 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. In the maintenance of a real system it should be possible to develop most of the remaining changes as dynamic updates, as will be discussed further in Section 8.1, which discusses solutions to the main problems encountered in converting K42 changes to dynamic updates.

Another desired property of the dynamic update model was that it should have an acceptably low performance impact, and should not constrain the usability or scalability of the system. The results of the performance measurements on the K42 prototype reported

in Chapter 6 showed that support for dynamic update imposed negligible performance impact on system throughput (less than 1% overhead), and that when updates were being applied, the performance impact was moderate (around 10% for a complex update). Hence, dynamic update functionality need not constrain system performance.

Chapter 7 has argued that the model could be extended beyond K42 to other modular operating systems. Although an implementation in Linux has been deferred to future work, it appears that the biggest limitation of such a system would be the extent of the kernel's modularity. Related to this, Section 8.2 compares and contrasts the modular approach taken in this research to other work in the area of dynamic update to operating systems.

Finally, Section 8.3 discusses general lessons for structuring an operating system to be dynamically updatable.

8.1 Common problems experienced with K42 updates

In the manual analysis described in Section 5.3, comments were recorded noting why a change couldn't be converted to a dynamic update. The problems preventing a change from being converted generally fell into one of the following categories:

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

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

8.1.1 Restructure of initialisation code

The most common single example of a static code problem is the kernel initialisation sequence. This code executes once only at boot time, 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 here 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 would be the same as the corresponding boot-time code.

A mechanism could be added to K42 that allows programmers to annotate initialisation functions in class header files. The annotated functions would then be called in a predictable order at boot time, or when the class was loaded as a dynamic update. This is similar to Linux's *initcall* mechanism [Woe06], 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, the K42 test system could be redesigned to allow test functions to be registered dynamically, enabling dynamic update as well as dramatically improving the source code.

8.1.2 Exception handlers and other non-modular code

Parts of the K42 kernel are not implemented as hot-swappable objects, and thus cannot be dynamically updated. 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 bug fixes and performance improvements in the CVS analysis.

In some cases it is possible to achieve a dynamic update to exception handlers by modifying other code. For example, one could update code 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 low-level 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, as described in Section 2.2.3, 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 [CCZ+06].

8.1.3 Dynamic tracing support

K42's tracing service [WR03] 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 dynamic update was not considered 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, the static enumeration could be changed to a dynamic structure, or a totally dynamic tracing service could be used [TM99, CSL04].

8.2 Comparison to related work

A number of other systems have directly attacked the problem of dynamic update to operating systems, all focusing on Linux. They have previously been described in Section 2.3, however it is now possible to revisit this related work and directly compare it with the results of this work.

8.2.1 Binary kernel patching: LUCOS and DynAMOS

In Section 2.3, two systems were described that aimed to support dynamic updates to the Linux kernel: LUCOS [CCZ+06] and DynAMOS [MR07]. The designers of these systems both avoided any modifications to the kernel's source code, by implementing dynamic update using binary code patching and a kernel module (in the case of LUCOS, the Xen virtual machine monitor was also required).

Compared to the modular approach developed in this work, the main advantage of these systems is that they do not require changes to the kernel source, and are able to apply updates to practically any kernel function. Furthermore, because they consist only of passive modules, they do not add any overhead to the system's base performance (although LUCOS requires virtualisation, which has its own significant overheads). However, these systems incur a significantly higher overhead when applying updates, resulting from their use of binary function patching as the underlying mechanism. For example, LUCOS incurs an exception and runs conversion functions on any write to an updated data structure, and DynAMOS adds significant overhead for every updated function due to function indirection.

LUCOS does not support any changes to function signatures. However, as the results in Chapter 5 showed, changes to interfaces are common in the maintenance of K42. To verify that this problem was not unique to K42, the author 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. Nevertheless, four of the seven versions examined included changes to the prototypes of non-inline kernel functions, confirming the prevalence of interface change in Linux's evolution, even in maintenance releases. Given these results, it seems questionable whether LUCOS offers sufficient support even for maintenance updates.

In conclusion, the comparison between the modular approach of this work and the binary function patching of LUCOS and DynAMOS 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 the changes that can be supported. As has already been discussed, the approach developed in this work is limited by the extent of modularity in the kernel, but where it can be used, it enables cheaper and simpler updates. A combination of the two approaches may be the best solution—using a system's modularity where it is available, and falling back to binary patching where it is not.

8.2.2 Automatic indirection: Ginseng

Ginseng [NHS+06] is a general-purpose dynamic update system for C programs, that operates by automatically inserting indirection on functions and data structures at compile-time, and uses a static analysis to determine safe points to apply updates. It was described in Sections 2.1.4 and 2.3.3. In attempting to apply Ginseng to the Linux kernel [NH06], the authors propose annotating the kernel's source code to help identify types that should have indirection added, and to denote transactions around which to apply updates. In this regard, they are taking an approach somewhere in between the binary patching approach, which makes no modifications at all to the kernel source, and the approach developed in this work, which modifies the kernel enough to add indirection, safe-point detection and state tracking features.

Although it is not yet clear if applying the Ginseng tool to the Linux kernel will be tractable, if it worked, the choice between the Ginseng approach and this work would represent a trade-off between the complexity of modifications required to the kernel's source code, and the performance of the resulting system. Because the approach developed by this work uses the modularity and indirection already present in the kernel, whereas Ginseng selectively adds indirection, the Ginseng-compiled Linux system could be expected to have higher overhead in the base case.

8.2.3 Checkpoint and migration of applications: AutoPod

As detailed in Section 2.3.4, AutoPod [PN05] is a kernel module and indirection layer for Linux that allows applications to be check-pointed while running on one version of a

kernel, and then restarted on a different version. By combining AutoPod with virtualisation, it is possible to achieve a dynamic update by starting a new virtual machine with the updated OS, then using AutoPod to checkpoint and migrate applications from the old OS to the updated version.

AutoPod represents an entirely different approach to the problem of dynamic update. Because it avoids significant modifications to the OS, and operates at the level of the relatively stable system call interface, it avoids most of the complex issues faced by systems that attempt to modify the OS kernel as it executes. Furthermore, it avoids the need to prepare updates separately from normal kernel development, although the AutoPod software must be ported to each new kernel version.

In an environment where virtualisation is already used for other reasons, the AutoPod approach offers compelling advantages over systems which modify the kernel as it runs. However, virtualisation generally imposes greater overheads than a standalone dynamic update system, so in other situations the simplicity of the AutoPod approach comes with a performance cost.

8.2.4 Distributed update: Upstart

Upstart [ALS03, Ajm04, ALS06] is a dynamic update framework for distributed systems, that operates 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 modular dynamic update as developed in this work and realised in K42. Instead of interposing on remote procedure calls between hosts, dynamic update in K42 interposes on method calls between objects, but many of the basic principles are similar. Upstart's transform functions are equivalent to K42's state transfer functions, simulation objects play the same role as adaptors, and scheduling objects are a generalisation of the choice between immediate and lazy update. It may even be feasible to use a similar centralised update service to distribute dynamic updates to clusters of hosts.

8.3 Lessons for building an updatable system

Having designed and implemented dynamic update for the K42 operating system and discussed the main issues facing an implementation of dynamic update for Linux or FreeBSD, it is now possible to draw some general lessons for structuring an OS to enable dynamic update. The OS designer who is developing a new system or restructuring an existing OS, and wishes to support dynamic update in a modular fashion should consider the issues that follow.

Most importantly, they should carefully *consider the modularity and structure of the system*. The dynamic update model proposed by this work requires modularity, and although it need not be fine-grained, such modularity also enables better adaptability and customisability of the system. Careful attention is required to selecting the appropriate module boundaries, so that the complexity of and frequency of changes to cross-module interfaces is minimised, because such changes are more difficult to support as dynamic updates.

Having selected the appropriate breakdown into modules, they should be careful to *respect module boundaries*, and *avoid data structures maintained by multiple modules*. If multiple modules need to access the same data structure, this could indicate that the wrong module boundaries have been selected. Furthermore, updates to such structures are more complex to apply.

Although not critical, the use of a *consistent module invocation mechanism* greatly simplifies the implementation of dynamic update. The alternative, which is seen in systems such as Linux and BSD that have evolved different function-pointer interfaces for each modular kernel API, requires that dynamic update be implemented separately for each module interface.

OS designers should seek to *minimise statically-bound code* that is located outside module boundaries. Some static code is inevitable, however by keeping such code simple, and relegating the complex algorithms and data structures more likely to evolve to modules, designers can greatly reduce the future complexity of updates to non-modular code.

Finally, designers should attempt to *use dynamic data structures* that are allocated and created at run-time rather than compile time. Dynamic structures are easier to update, because a new version of the structure may be allocated separately from the old version, whereas if a static structure is updated its size cannot increase, because its space is pre-allocated at compile time.

These guidelines are not absolute requirements for a system to support dynamic update (those were established in Section 3.1), however they show how to make it as simple as possible. Building a system with such modularity and dynamic structures has many other advantages beyond enabling dynamic update. Besides the well-established software engineering advantages of modularity, including better maintainability and better extensibility, it also enables more customisable and adaptable behaviour of the OS. Although a modular structure incurs extra overheads over a monolithic system, this work has shown that the costs can be kept low. Furthermore, as the K42 experience has shown, a modular structure can dramatically improve the multi-processor scalability of an OS [GKA+99, AAB+05, KAR+06].

9 Conclusions

Reboots for critical OS patches and updates are a major problem for a large group of system users and administrators, who are forced to trade the cost of the downtime required for a restart against the risk of remaining vulnerable to whatever flaw the update corrects. To avoid this problem, the aim of this research has been to develop a model for dynamic update of operating systems, using modularity to enable staged updates to the OS.

The approach taken for this research has been based on a case-study implementation of a modular dynamic update system for the K42 OS. This prototype is able to apply updates to kernel code and data structures, even when the interfaces between modules change. To enable changes to data structures with many instances, updates may be applied lazily. Measurements of the update mechanism have shown that, in contrast to the inherent costs of non-modular approaches, dynamic update features can have low overhead and need not affect system performance after updates are applied. Furthermore, the applicability of the update system has been assessed by an analysis of the K42 revision history, which found that at least 79% of maintenance changes could directly be converted to dynamic updates, and, as was discussed in the previous chapter, the proportion would be closer to 100% if the changes were developed for dynamic update. Finally, as was argued in Chapter 7, the model applies to other commodity systems such as Linux and FreeBSD, that although structured modularly, are not strictly object-oriented like K42.

In conclusion, the development of dynamic update features for modular operating system kernels is both possible and practical, and need not constrain system performance. Furthermore, the modular approach is appropriate for maintenance changes such as bug fixes. These are the most critical class of updates, and those featured in the motivating problem.

Finally, the implication of this work is that given a sufficiently-modular operating system kernel, although some changes to the system are required, the existing modularity and indirection present within the system can be used to implement dynamic update features with significantly better performance characteristics than a low-level non-modular approach. The use of modularity to isolate and apply updates should also make it easier to develop and reason about updates. The designer of an operating system would be advised to carefully consider the modularity and structure of the system, and to use a consistent module invocation mechanism, to simplify the implementation of dynamic update.

9.1 Future work

Although this research has shown that dynamic update to operating system code is possible, more work is required to develop a complete and usable system. This section describes some promising areas for future work.

9.1.1 Validate Linux implementation

As the approach of this work 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 such as Linux, as was described in Chapter 7.

9.1.2 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 the examples in Section 4.3 have shown, it is possible to dynamically update a system using this design, the ideal update system would also automate update preparation from source code changes.

This is not a problem that this work has addressed, but it is also not unique to operating systems. Other work in the dynamic update field has developed tools to ease the construction of dynamic patches [ABB+05, NHS+06] and investigated the semi-automatic creation of state transformers [Lee83, VB03, NHS+06]. It would also be possible to generate common adaptor objects from source code analysis.

9.1.3 Arbitrary interface changes

As currently designed, the dynamic update system cannot support interface changes in which the changes are not backwards-compatible, and thus 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 it would be desirable to support all dynamic updates.

As explained previously, support for arbitrary interface changes requires blocking all objects or modules affected by a change, potentially the whole kernel. Kernel events could be blocked at a lower level, such as exception handlers or an 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.

There is a possible optimisation for this suggested technique. Many data structures are accessed infrequently, so when an update was loaded the system could be allowed to continue executing while all state was optimistically transformed to new versions. Then, during the critical blocked period, those data structures that had been accessed after their state was transformed would be detected, and only their transfer functions would be re-invoked, significantly reducing the critical blocked period.

However, for practical applications of dynamic update, requiring all interface changes to be backwards-compatible appears sufficient. This model allows new interfaces to be defined by updates, as long as existing interfaces are still supported. This requirement also mirrors the approach taken by component systems such as COM [COM95], where interfaces cannot be changed once defined.

9.1.4 Updates outside the kernel

Many operating system updates also change user-level code such as system libraries. Furthermore, in K42 and other systems some system functionality is implemented by user-level servers. There is nothing preventing the dynamic update mechanism from operating at user-level, or in server processes. However, depending upon the structure of the relevant libraries or applications, general-purpose dynamic update systems, as described in Section 2.1, may be more suitable or practical.

Dynamic updates that affect both sides of the user-kernel boundary, such as changes to the system call interface, will require special support. It will be necessary to ensure that the kernel is updated before any user process. Although in a production system any changes would usually be backwards compatible, specialised adaptors could be used between the updated kernel interface and un-updated user processes.

9.1.5 Rolling back failed patches

In some cases, an administrator may wish to revert or roll back an update after it has been applied. This can be expected to occur rarely, because maintenance updates should 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.

Acknowledgements

I've always found acknowledgement sections in theses a little trite and ostentatious, appearing as they do at the beginning of the document, and stealing the reader's attention from the material that follows. So, much like a regular paper or article, I've elected to place them here as part of the back-matter, to signify that while there is much to acknowledge, the reader is excused from reading them unless he or she is genuinely interested.

First, I'd like to thank the members of the former K42 group at IBM Research, who gave me both a stimulating environment in which to conduct an internship, the flexibility to choose a research topic which was sizable enough to lead to the work presented here, and continuing support afterwards. In particular, Jonathan Appavoo helped to flesh out my ideas and provided plenty of his own over the course of many late-night telephone calls. Bob Wisniewski also helped to develop the ideas, and did his best to critique my writing and help out with papers when deadlines were tight. Dilma Da Silva and Orran Krieger also helped develop the work. Bryan Rosenburg can read the entrails of a K42 crash like no-one else, and on several occasions rescued me from my own bugs by asking just the right questions. Jeremy Kerr of IBM OzLabs took on the implementation of the module loader, helped me to understand some of the issues with Linux, and assisted me in converting his unforced file sync patch (from Section 4.3.4) into a dynamic update. Raymond Fingas from the University of Toronto developed the arbiter objects, on which the implementations of lazy update and adaptors were based.

Over the past few years I've had the privilege of meeting and discussing my work with a variety of people. Iulian Neamtiu and Eno Thereska critiqued paper drafts and had helpful suggestions. The attendees of both the First EuroSys Doctoral Workshop and First EuroSys Authoring Workshop provided useful feedback and a forum in which to present early work. The anonymous and not-so-anonymous members of conference program committees and the examiners of this thesis have also provided many helpful comments.

Back at home, I need to thank my supervisor Gernot Heiser for stimulating my interest in operating systems. Despite his many hats, Gernot was always available to provide advice or read drafts, often on short notice. To all the past and present members of the ever-growing KEG lab, thanks for providing so many avenues of procrastination. Although it's high time I left, I will miss the camaraderie.

Finally, to my parents and my biggest supporter, Sarah, many thanks for getting me through this.

References

[AAB+05] Jonathan Appavoo, Marc Auslander, Maria Butrico, Dilma M. Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Experiences with K42, an open-source, Linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–440, April 2005.

[AADS+03] Jonathan Appavoo, Marc Auslander, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, Ben Gamsa, Reza Azimi, Raymond Fingas, Adrian Tam, and David Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. IBM Research Report RC22863, IBM Research, July 2003.

[AAE+03] Jonathan Appavoo, Marc Auslander, David Edelsohn, Dilma Da Silva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Proceedings of the 2003 Annual USENIX Technical Conference, FREENIX Track*, pages 323–336, San Antonio, TX, USA, June 2003.

[ABB+05] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online patches and updates for security. In *Proceedings of the 14th USENIX Security Symposium*, pages 287–302, Baltimore, MD, USA, August 2005.

[ABG+04] Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund, Toke Eskildsen, Klaus Marius Hansen, and Mads Torgersen. Design, implementation, and evaluation of the Resilient Smalltalk embedded platform. In *Proceedings of the 12th European Smalltalk User Group Conference*, Köthen, Germany, September 2004.

[AHS+02] Jonathan Appavoo, Kevin Hui, Michael Stumm, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Craig A. N. Soules. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of the ACM SIGSOFT Workshop on Self-Healing Systems*, pages 3–8, Charleston, SC, USA, November 2002.

[AHS+03] Jonathan Appavoo, Kevin Hui, Craig A. N. Soules, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, Marc Auslander, David Edelsohn, Ben Gamsa, Gregory R. Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm, and Jimi Xenidis. Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76, 2003.

[Ajm04] Sameer Ajmani. *Automatic Software Upgrades for Distributed Systems*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, September 2004. Also as Technical Report MIT-LCS-TR-1012.

[ALS03] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 43–48, May 2003.

[ALS06] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of the 20th European Conference on Object Oriented Programming*, July 2006.

[App05] Jonathan Appavoo. *Clustered Objects*. PhD thesis, Department of Computer Science, University of Toronto, 2005.

[AVW+96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*, chapter 9, pages 121–123. Prentice Hall, 2nd edition, 1996.

[BA05] Andrew Baumann and Jonathan Appavoo. Improving dynamic update for operating systems. In *Proceedings of the 20th ACM Symposium on OS Principles, Work-in-Progress Session*, Brighton, UK, October 2005.

[BADS+04] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Improving operating system availability with dynamic update. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, USA, October 2004.

[BAW+07] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proceedings of the 2007 Annual USENIX Technical Conference*, pages 337–350, Santa Clara, CA, USA, June 2007.

[BC02] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, 2002.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on OS Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[BHA+05] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 Annual USENIX Technical Conference*, pages 279–291, Anaheim, CA, USA, April 2005.

[BHS+03] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*, Warsaw, Poland, April 2003.

[BKA+05] Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, Australia, April 2005.

[Blo83] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1983. Also as Technical Report MIT-LCS-TR-303.

[BLS+03] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Proceedings of the ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 403–417, Anaheim, CA, USA, October 2003.

[CAK+96] Crispin Cowan, Tito Audrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, Annapolis, MD, USA, May 1996.

[CCZ+06] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd USENIX Symposium on Virtual Execution Environments*, pages 35–44, Ottawa, Canada, June 2006.

[CG05] Ludmila Cherkasova and Rob Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proceedings of the*

2005 Annual USENIX Technical Conference, pages 387–390, Anaheim, CA, USA, April 2005.

[CKF+04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot—a technique for cheap recovery. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, December 2004.

[COM95] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*, 1995. http://www.microsoft.com/com.

[Coo80] Robert P. Cook. *MOD—a language for distributed programming. *IEEE Transactions on Software Engineering*, 6(6):563–571, November 1980.

[CSC72] Fernando J. Corbató, Jerome H. Saltzer, and Charles T. Clingen. Multics—the first seven years. In *Proceedings of the Spring Joint Computer Conference*, 1972.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 Annual USENIX Technical Conference*, pages 15–28, Boston, MA, USA, June 2004.

[DD68] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, May 1968.

[dGdS99] Juan-Mariano de Goyeneche and Elena Apolinario Fernández de Sousa. Loadable kernel modules. *IEEE Software*, 16(1):65–71, January 1999.

[DH01] Pär Danielsson and Torbjörn Hultén. JDRUMS: Java distributed run-time updating management system. Master's thesis, Department of Mathematics, Statistics and Computer Science, Växjö University, S-35252 Växjö, Sweden, January 2001.

[DPM02] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Computing Surveys*, 34:450–468, December 2002.

[Dra93] Richard Draves. The case for run-time replaceable kernel modules. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 160–164, Napa, CA, USA, October 1993.

[DSKW+06] Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, Amos Waterland, David Tam, and Andrew Baumann. K42: an infrastructure for operating system research. *Operating Systems Review*, 40(2):34–42, April 2006.

[Dug01] Dominic Duggan. Type-based hot swapping of running modules. In *Proceedings of the International Conference on Functional Programming*, pages 62–73, Florence, Italy, September 2001.

[Fab76] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd ICSE*, pages 470–476, San Francisco, CA, USA, 1976.

[FGC+05] Marc E. Fiuczynski, Robert Grimm, Yvonne Coady, and David Walker. `patch (1)` considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, 2005.

[Fin06] Raymond Fingas. Interposing on calls in the K42 operating system. M.A.Sc. thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto, Canada, 2006.

[Fre06] FreeBSD Documentation Project. *FreeBSD Architecture Handbook*, 2006. http://www.freebsd.org/doc/en/books/arch-handbook/.

[Gam99] Ben Gamsa. *Tornado: Maximizing Locality and Concurrency in a Shared-Memory Multiprocessor Operating System*. PhD thesis, Department of Computer Science, University of Toronto, 1999.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GIL78] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. Dynamic restructuring in an experimental operating system. In *Proceedings of the 3rd ICSE*, pages 295–304, Atlanta, GA, USA, 1978.

[GJ93] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software: Practice and Experience*, 23(9):949–964, September 1993.

[GJB96] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.

[GJP+00] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The Sawmill multiserver approach. In *Proceedings of the 9th SIGOPS European Workshop*, pages 109–114, Kolding, Denmark, 2000. ACM Press.

[GKA+99] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, New Orleans, LA, USA, February 1999. USENIX.

[GKS94] Ben Gamsa, Orran Krieger, and Michael Stumm. Optimizing IPC performance for shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 208–211, 1994.

[GKW97] Stephen Gilmore, Dilsun Kírlí, and Christopher Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, Department of Computer Science, The University of Edinburgh, December 1997.

[GPR+98] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the 1998 Annual USENIX Technical Conference*, pages 39–52, New Orleans, LA, USA, 1998.

[Gra03] Patrick Gray. Experts question Windows patch policy. ZDNet News, November 2003.

[GSB+99] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 Annual USENIX Technical Conference*, pages 267–282, Monterey, CA, USA, June 1999.

[Gup94] Deepak Gupta. *On-Line Software Version Change*. PhD thesis, Department of Computer and Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[Gus01] Jens Gustavsson. Dynamic updating of computer programs—proposed improvements to the JDrums updating system. In *Proceedings of the 3rd Conference on Computer Science and Systems Engineering in Linköping*, pages 243–248, Norrköping, Sweden, March 2001.

[HAW+01] Kevin Hui, Jonathan Appavoo, Robert Wisniewski, Marc Auslander, David Edelsohn, Ben Gamsa, Orran Krieger, Bryan Rosenburg, and Michael Stumm. Supporting hot-swappable components for system software. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001.

[HBB+98] Hermann Härtig, Robert Baumgartl, Martin Borriss, Claude-Joachim Hamann, Michael Hohmuth, Frank Mehnert, Lars Reuther, Sebastian

Schönberg, and Jean Wolter. DROPS—OS support for distributed multimedia applications. In *Proceedings of the 8th SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[HBG+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *Operating Systems Review*, 40(3):80–89, July 2006.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[HF98] Johannes Helander and Alessandro Forin. MMLite: A highly componentized system architecture. In *Proceedings of the 8th SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[HG98] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes—a lightweight mechanism to update code in a running program. In *Proceedings of the 1998 Annual USENIX Technical Conference*, pages 65–76, June 1998.

[Hic01] Michael Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[HKS+05] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications and Services*, pages 163–176, Seattle, WA, USA, November 2005.

[HN05] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, November 2005.

[Hon81] Honeywell Information Systems. *Introduction to Programming on Multics*, July 1981. Order number AG90-03.

[HW96] Steffen Hauptmann and Josef Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 70–80, Annapolis, MD, USA, May 1996.

[IBM02a] IBM K42 Team. *K42 Overview*, August 2002. Available from http://www.research.ibm.com/K42/.

[IBM02b] IBM K42 Team. *Memory Management in K42*, August 2002. Available from http://www.research.ibm.com/K42/.

[IBM02c] IBM K42 Team. *Utilizing Linux Kernel Components in K42*, August 2002. Available from http://www.research.ibm.com/K42/.

[KAR+06] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *Proceedings of the EuroSys Conference*, pages 133–145, Leuven, Belgium, April 2006.

[Kle86] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 238–247, Atlanta, GA, USA, June 1986.

[Lav05] Dru Lavigne. FreeBSD: An open source alternative to Linux. FreeBSD Project, 2005. http://www.freebsd.org/doc/en/articles/linux-comparison/.

[Lee83] Insup Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, University of Wisconsin-Madison, May 1983.

[LSS04] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines: Enabling general, single-node, online maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–223, October 2004.

[MBK+96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[Min02] Ronald G. Minnich. A dynamic kernel modifier for Linux. In *Proceedings of the 3rd Los Alamos Computer Science Institute Symposium*, Santa Fe, NM, USA, October 2002.

[MLR+04] Keir B. Mierle, Kevin Laven, Sam T. Roweis, and Greg V. Wilson. CVS data extraction and analysis: A case study. Technical Report UTML TR 2004-002, Dept of Computer Science, University of Toronto, September 2004.

[Moo01] Richard J. Moore. A universal dynamic trace for Linux and other operating systems. In *Proceedings of the 2001 Annual USENIX Technical Conference, FREENIX Track*, Boston, MA, USA, June 2001.

[Moo04] Jim J. Moore. High performance dynamically updateable software architecture. *Research Disclosure Journal*, 481010, May 2004.

[MR07] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the EuroSys Conference*, pages 327–340, Lisbon, Portugal, March 2007.

[MS98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, NV, USA, October 1998.

[MSA+02] Paul E. McKenney, Dipankar Sarma, Andrea Arcangelli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[NFH05] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2nd International Workshop on Mining Software Repositories*, pages 2–6, Saint Louis, MO, USA, May 2005.

[NH06] Iulian Neamtiu and Michael Hicks. Dynamic software updating for the Linux kernel. USENIX Symposium on Operating Systems Design and Implementation, Work-in-Progress Session, November 2006. Seattle, WA, USA.

[NHS+06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[Org72] Elliott I. Organick. *The Multics System: An Examination of its Structure*. MIT Press, 1972.

[OSS+02] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 361–376, Boston, MA, December 2002.

[PAB+95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: streamlining a commercial operating

system. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 314–321, Copper Mountain, CO, USA, 1995.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PLM06] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *Proceedings of the EuroSys Conference*, pages 59–71, Leuven, Belgium, April 2006.

[PN05] Shaya Potter and Jason Nieh. Reducing downtime due to system maintenance and upgrades. In *Proceedings of the 19th USENIX Large Installation System Administration Conference*, pages 47–62, San Diego, CA, USA, December 2005.

[PW93] Calton Pu and Jonathan Walpole. A study of dynamic optimization techniques: Lessons and directions in kernel design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, 1993.

[RA00] Tobias Ritzau and Jesper Andersson. Dynamic deployment of Java applications. In *Proceedings of the Java for Embedded Systems Workshop*, London, UK, May 2000.

[Rei00] Andrew Reiter. Dynamic kernel linker (KLD) facility programming tutorial. *Daemon News*, October 2000.

[Res03] Eric Rescorla. Security holes... who cares? In *Proceedings of the 12th USENIX Security Symposium*, pages 75–90, Washington, DC, USA, August 2003.

[Rit84] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[Rub98] Alessandro Rubini. *Linux Device Drivers*. O'Reilly & Associates, Inc, 1998.

[SAB+04] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, December 2004.

[SAH+03] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the 2003 Annual*

USENIX Technical Conference, pages 141–154, San Antonio, TX, USA, 2003.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on OS Principles*, Bolton Landing (Lake George), New York, USA, October 2003.

[SC05] Don Stewart and Manuel M. T. Chakravarty. Dynamic applications from the ground up. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, Tallinn, Estonia, September 2005.

[Sch71] Roger R. Schell. *Dynamic Reconfiguration in a Modular Computer System*. PhD thesis, Project MAC, Massachusetts Institute of Technology, June 1971. Also as Technical Report MIT-LCS-TR-086.

[SF93] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2):53–65, March 1993.

[SGI94] Silicon Graphics, Inc. *IRIX 5.3 Device Driver Programming Guide*, November 1994. Document number 007-0911-050.

[SHB+05] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Long Beach, CA, USA, January 2005.

[Sun91] Sun Microsystems Inc. *SunOS 4.1.3 Reference Manual*, 1991.

[Sun01] Sun Microsystems Inc. *Solaris Live Upgrade 2.0 Guide*, October 2001. Document number 806-7933-10.

[Sun02] Sun Microsystems Inc. *The Java HotSpot Virtual Machine, v1.4.1, d2*, September 2002. Technical white paper. Available from http://java.sun.com/products/hotspot/.

[TM99] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 117–130, New Orleans, LA, USA, February 1999.

[VB03] Yves Vanderwoude and Yolande Berbers. A meta-model driven methodology for state transfer in component-oriented systems. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*, Warsaw, Poland, April 2003.

[VH96] Alistair C. Veitch and Norman C. Hutchinson. Kea—a dynamically extensible and configurable operating system kernel. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, pages 123–129, Annapolis, MD, USA, May 1996.

[VH98] Alistair C. Veitch and Norman C. Hutchinson. Dynamic service reconfiguration and migration in the Kea kernel. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 156–163, Annapolis, MD, USA, May 1998.

[WKG00] Chris Walton, Dilsun Kírlí, and Stephen Gilmore. An abstract machine model of dynamic module replacement. *Future Generation Computer Systems*, 16(7):793–808, May 2000.

[Woe06] Trevor Woerner. Understanding the Linux kernel initcall mechanism, October 2006. http://geek.vtnet.ca/doc/initcall/.

[WR03] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proceedings of the 17th International Conference on Supercomputing*, Phoenix, AZ, USA, November 2003.

[WRA05] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on OS Principles*, Brighton, UK, October 2005.

[ZW04] Thomas Zimmermann and Peter Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, Edinburgh, UK, May 2004.