Design and Implementation of K42's Dynamic Clustered Object Switching Mechanism

by

Kevin Hui

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Design and Implementation of K42's Dynamic Clustered Object Switching Mechanism

Kevin Hui

Master of Science

Graduate Department of Computer Science

University of Toronto

2000

Recent research efforts have investigated customizable operating systems, where the implementation of operating system services can be chosen to meet an application's performance or functionality requirements. This dissertation investigates the potential benefits of allowing the customization to be changed, on-the-fly, while the service is in use. By using a prototype implementation of the dynamic object switching layer in the K42 operating system, we explore the costs and benefits associated with dynamic customization. As an example, we showed how K42 can switch a (per-file) page cache from a centralized implementation to one distributed across the processors of a multiprocessor in order to adapt to changing access patterns. The ability to customize on-the-fly allows the implementation of a service to match the instantaneous demands on the service, avoiding the need to comprise a complex, catch-all implementation. It also facilitates live-swapping of system components in mission-critical systems where downtime is undesirable.

Acknowledgements

I would like to thank all the members of the K42 research group at the IBM T. J. Watson Research Center, who have been excellent sources of ideas and answers: Mark Auslander, David Edelsohn, Orran Krieger, Craig MacDonald, Bryan Rosenburg, Rick Simpson, Robert Wisniewski, and Jimi Xenidis. I would especially like to thank Robert Wisniewski, who acted as my mentor during my visit at the IBM lab, for his helpful supervision and invaluable advice. I would also like to thank K42 Toronto researchers Ben Gamsa and Jonathan Appavoo for their K42/Tornado knowledge and other advice in general. Last but certainly not least, I thank my thesis advisor, Professor Michael Stumm, for giving me the opportunity to work on K42 at the IBM lab, and for all the guidance he provided.

Contents

1 Introduction **1**

1.1 The K42 Operating System 2

1.2 Dynamic Customization in K42 4

1.3 Dissertation Outline 6

2 Related Work **7**

3 K42 Operating System **11**

3.1 K42 Object Infrastructure 12

3.2 Building-Block Composition 13

3.3 Clustered Objects 18

3.3.1 Overview 18

3.3.2 Implementation 20

3.3.3 Generation Count 24

3.4 Summary 26

4 Dynamic Object Switching **28**

4.1 Background 29

4.1.1 Hybrid object 30

4.1.2 K42's dynamic switching design goals 32

4.2 K42's approach to dynamic switching 32

4.3 Implementation overview 33

4.4 Implementation details 40

4.4.1 The Mediator 40

4.4.2 Deadlock Avoidance 47

4.4.3 Multiprocessor implementation issues 49

4.4.4 Data Transfer 52

4.4.5 Implementation Status 57

4.5 Summary 57

5 Performance **59**

5.1 Experimental Framework 59

5.2 Dynamic Switching Costs 61

5.2.1 Fixed costs 62

5.2.2 Variable costs 62

5.3 Experimental Results 63

5.3.1 Description of the Counter Objects 63

5.3.2 Counter Experiment 64

5.3.3 Results of Counter Experiments 65

5.3.4 Description of the K42 Objects 67

5.3.5 Page Fault Experiment 68

5.3.6 Results of Page Fault Experiment 69

5.4 Cost-Benefit Tradeoffs in Dynamic Switching 69

5.5 Summary 71

6 Concluding Remarks **73**

6.1 Future Work 75

Bibliography **76**

Chapter 1

Introduction

There has been much recent research on customizable operating systems. In a customizable operating system, virtual resources, such as files and virtual memory regions, can be tailored to optimize for a specific usage pattern on a per-application basis. For example, an application that can benefit from sequential file pre-fetching can inform the operating system of this and the system can then provide the application with a file object with such a pre-fetching policy. Examples of customizable operating systems with such a capability include Cache Kernel, Choices, Exokernel, SPIN, and VINO. In these systems, however, the customization is generally done statically at resource creation time, and the resource implementation created will stay unchanged for the lifetime of the resource. While more flexible than traditional operating systems (where there is only a single generic implementation of a resource for all varieties of applications), these systems lack the ability to dynamically adapt to changes in program phases where the access patterns of the resource change dramatically. Also, for mission-critical systems that require constant up-time, the ability to change the implementation of a system resource (for reasons such as feature upgrades and bug-fixes), without having to halt the system or application, can be invaluable. This dissertation explores the possibility of performing dynamic, on-the-fly switching of operating system resource implementations, even if the resource has already been created and is being used actively and concurrently in a multiprocessor system.

1.1 The K42 Operating System

K42[1] is a new customizable operating system we are developing jointly with IBM research for 64-bit shared-memory multiprocessors. Three important goals of the system are

Scalability: to *scale up* to support very large systems (hundreds of processors) and to support applications that utilize the entire system, while also *scaling down* to support efficiently 1) the small-scale multiprocessors we expect to be ubiquitous in the near future, and 2) sequential and small-scale parallel applications on a large-scale multiprocessor;

Maintainability, extensibility, and portability: so as to 1) avoid the high maintenance costs of existing operating systems, 2) enable the system to be extended easily to support new types of applications and integrate new research ideas; 3) allow the system to be easily ported to new hardware platforms, and 4) optimize performance by exploiting hardware specific features;

Application-specified customizability: to allow subsystems (e.g., data bases, web servers, JVMs) and scientific applications to customize the operating system management of the resources they use.

To achieve these goals, K42 is implemented using an object-oriented structuring technique called *building-block composition* [2, 16]. Each virtual resource (i.e. virtual memory region, network connection, file, process, etc.) is implemented by a different composition (or set) of objects, allowing resource management policies and implementations to be controlled on a per virtual resource (and thus per application) basis. This allows, for example, every open file to have a different pre-fetching policy, every memory region to have a different

[1]K42 was originally named Kitchawan.

page size, and every process to have a different exception handling policy. We refer to the overall implementation of a virtual resource as a building-block composition, and to the individual objects in the composition as building blocks.

Building-block composition is a natural way to structure code for multiprocessors. Since each resource is implemented using a different instance of building blocks, independent requests to different resources can be serviced in parallel. In K42 there are no global data structures that need to be traversed and no global locks that need to be acquired.[2]

Some building blocks, such as those used for a shared file or the *Process* building block used for a parallel program, may be widely shared across a large multiprocessor. To implement these widely shared building blocks efficiently, the concept of *clustered objects* [1, 12, 29] has been developed. A clustered object building block is one that can be partitioned into *representative (rep) objects*, where independent requests on different processors are (in the common case) handled by different representatives of the object. A clustered object is like any other building block as far as its clients are concerned, and the implementation can be chosen at instantiation while maintaining the same interface. Clustered objects provide the additional flexibility to modify the level of distribution. For example, a *Process* building block for a parallel program is implemented in the K42 kernel using a rep for each processor on which the application runs — many common operations (e.g., in-core page faults) are then handled by the single, local rep of the Process without requiring any communication with the other reps. The clustered object infrastructure allows the parallel Process object to export the same interface as the Process object designed to run on a uniprocessor, and the distribution and locations of the reps are transparent to the clients of the clustered object.

Customizability is achieved in K42 by letting an application specify which building blocks the operating system should use to implement access to the resources used by the application. Moreover, with the infrastructure described in this dissertation, the building

[2]No other operating system we are aware of has this characteristic.

blocks used to implement a resource can be changed, on the fly, even if the resource is actively being accessed. For example, an application can direct the operating system to change the *Process* building block from a centralized implementation (with a single shared rep) to a distributed one (with a rep per processor).

1.2 Dynamic Customization in K42

Building blocks provide tremendous flexibility in allowing K42 to be customized for an application. As other work in customizable systems demonstrates [3, 4, 6, 8, 9, 26], this flexibility can translate into significant performance gains. Often times though, when a resource is first accessed, it is not clear, especially from the operating system's perspective, what its request pattern will be. Also, an application's use of operating system resources may change over time as the application goes through different phases. While customization via building blocks increases performance, there are common scenarios where additional performance gains can be obtained if the customization can be dynamic. For the different kinds of applications an operating system needs to support, a single implementation of building blocks will not perform as well as when the operating system can dynamically switch the building blocks implementing a given resource when the behavior changes. This is because the system does not know *a priori* the access patterns for an resource, or because the access pattern changes significantly over time.

The ability to customize on the fly allows K42 to optimize the building blocks used to meet the instantaneous demands on a resource. We can have many simple implementations of building blocks, each specialized to meet particular needs, avoiding the need to develop complex compromise building blocks that meet a variety of different needs. This has important implications for achieving the goals of K42 as described above:

- simple (non-scalable) building blocks that pay no performance overhead for scalability can be used for sequential applications, allowing sequential applications to run more efficiently on a large multiprocessor,

- we can avoid the need to compile a separate version of the system for uniprocessors, since the dynamic customization captures the optimizations that a separate build captures statically (this avoids cluttering up the code with `#ifdef` statements),

- we can avoid the performance overhead of conditional branches needed in general-purpose implementations,

- there are fewer complex code paths making the system more maintainable,

- building blocks can be upgraded, updated, or replaced on the fly without needing to bring down long-running applications or the operating system, and

- researchers can easily introduce new special purpose building blocks without affecting other applications.

Allowing building-block compositions to be modified on the fly introduces a number of challenging problems. For example, we need to ensure that requests being serviced by the object during the switch are handled properly, producing the correct result. Also, we want the facility to be generally applicable to all building blocks, so it has to work generically over all interfaces. Further, while being a general infrastructure, we do not want to impose any overhead on building blocks that are not currently using the facility. The infrastructure we developed and describe in this dissertation addresses these problems, making it possible to add dynamic customizability to new resources with little programming effort, and acceptable overhead.

Having building blocks dynamically switch implementations could either be initiated pro-actively in response to a request from the application, or reactively based on continual performance monitoring done by the operating system. The dynamic switching mechanism we implemented in K42 replaces one building block instance with another, thereby allowing switches in internal data representation and distribution. The switching of objects may occur while the original object is processing requests. The switching mechanism correctly handles in-flight requests to the objects involved in the switch.

1.3 Dissertation Outline

This dissertation focuses on the implementation of the dynamic switching *mechanism* within the K42 operating system environment, not on the *policy* associated with switch initiation. The rest of the dissertation is organized as follows. Chapter 2 describes other customizable operating system work. Chapter 3 gives an overview of the K42 operating system and its approach to customizability. It also describes the building block and clustered object infrastructures. Chapter 4 describes the implementation of K42's dynamic switching mechanism and the tradeoffs we faced in designing this mechanism. Chapter 5 shows the performance advantages of dynamic switching over static customizability, both on a sample object as well as on K42 memory management objects. It also analyzes the programming and performance overhead of applying dynamic switching. Chapter 6 provides a summary of what we learned during the implementation, summarizes our mechanism and results, and describes possible future work.

Chapter 2

Related Work

A number of research operating system projects have explored customizability. Recent research in extensible and customizable operating systems includes SPIN [3, 22], VINO [26, 27], Exokernel [8], Fluke [9], L4 [18], Cache Kernel [6], Choices [4], Nemesis [17], Scout [20], and K42 [2].

Many of the above operating systems achieve customizability by extensibility. The Exokernel, Fluke, L4, and Cache Kernel allow for customizability by having the kernel redirect hardware events to external address spaces where they can be serviced in a customized way on a per-application basis.

Fluke and Cache Kernel employ a virtual machine model, where each application can have a different operating system running in a virtualized version of the hardware. Fluke proposes a recursive virtual machine model, in which layers of virtualized hardware, or virtual machines, stack up to offer customizability and extensibility. An application can run on any one of such virtual machine layers, and the application itself can be another layer of virtual machine for applications that require further-customized operating system services. Cache Kernel treats the operating system kernel as a set of cacheable objects; there can be multiple instances of different kernels coexisting in the system. The proper kernel component is faulted in and cached in memory when used by a particular application. To one degree or another, these approaches suffer from the overhead required to cross address space boundaries and they make the sharing of resources between applications desiring different extensions difficult. For example, Fluke interposes a layer of virtual machine implementing a new policy, say to customize the physical memory page replacement policy, and applications requiring the new policy will need to be run on top of this new layer of virtual machine, inducing extra cross-address-space invocations. In Cache Kernel, extra context switches need to be performed to fault in the appropriate customized kernel component. Also, if two customized versions of the file cache are caching the same file, it is more difficult to use the same memory to cache the file if they are implemented in two different address spaces.

SPIN and VINO achieve customizability by allowing user code, compiled from a safe language, to be downloaded into the kernel. SPIN employs an event mechanism to dispatch downloaded extensions [22]. Predicate functions, or guards, are evaluated in the dispatch path to decide which extension to execute. VINO allows extensions, or grafts, to be installed in system graft points. Both systems, while customizable, suffer from performance overheads resulting from predicate evaluation associated with the extensions. The larger the number of applications that require customization, the higher the overhead [26].

The systems described above all achieve customizability through extensibility, i.e., customized code is loaded into the system. In contrast, K42's building-block composition separates customizability and extensibility. Customizability is achieved by allowing applications to compose building blocks from a set of existing components to manage a resource. There is no performance degradation due to downloading or verification of the components. Also, since each application independently chooses a building-block composition appropriate for its requests to a resource (e.g., process object), the system provides customizability without affecting other applications building-block composition for their view of the resource. For example, one application can use a single, shared representation of a process object while another application can use a distributed representation. Extensibility is achieved by allowing trusted parties to write new objects that extend the functionality of the system. The separation leads to lower overhead [2]. As with other composable object-oriented systems [4, 20, 6], K42 has the benefit of better maintainability.

K42's solution to customizability can be considered to be much more conservative than the techniques chosen by other research groups. In many ways, building-block composition is not much different from other object-oriented techniques [10] and can be viewed as a specific realization of the Framework approach from the Choices operating system [4, 14]. These object-oriented techniques have already successfully been applied in commercial operating systems, for example with the Unix Vnode interface [30] and Streams facility [28]. We build on this previous work, taking advantage of its strengths with respect to maintainability, extending it to all components of the operating system, and, for customizability, providing a powerful mechanism for applications to control the objects used. Overall, we believe that building-block composition is simpler to use than the other more radical techniques, and that it results in better maintainability and performance because it does not require the system designer to deal with many of the complexity and security problems the other approaches must face. While similar challenges may still exist in our model, they are not the concern of the *users* of the building blocks, but the building-block *implementors*.

The ability to dynamically optimize data structures and policies used by the system, in conjunction with a customizable system, can improve performance of an application considerably. VINO leverages its extension mechanism to install performance monitoring grafts to collect performance data for both online and off-line processing [27]. The grafting infrastructure is also used to facilitate the development of possible system adaptations, but it does not provide a generic way to handle on-the-fly switching of object implementations at the interface level. The Synthesis operating system [24] features dynamic code optimization using an on-line kernel code generator to reduce the latency of common kernel functions. While effective when applied, this optimization requires hand-coded optimizations written in low-level machine language, which limits its applicability. Both of these techniques could be applied to K42 code, but the thrust of K42's dynamic customization is wider.

Dynamic linking introduces the ability to allow new code modules to be added to a running system, improving the customizability of a program. While useful, it can only be applied at the granularity of modules, lacking native support for the C++ language. Microsoft's component object model (COM) [21] allows coexistence of different component objects implementing the same interface, and uses dynamic linking to load in new modules when an instance of a new component implementation is created. On the Java front, the class loader system can be extended to support dynamically loadable classes also. A more lightweight dynamic C++ object system has been proposed, implementing dynamically loadable C++ objects that support versioning [13]. These facilities solve the problem of updating future object instances, but they are incapable of allowing already instantiated, long-running system objects to change their implementations dynamically, hence limiting their ability to perform live-swapping of such objects.

The Synthetix work [25] is closest in spirit to our own, supporting on-the-fly customization of the system in response to changing access patterns. However, they assume the code to be switched can only be called by a single thread at a time, that only a single function is to be switched, and that no data translation or object switching is required. In contrast to the method we propose here, they also pay overhead on every function call, even when no switching is occurring.

To the best of our knowledge, none of these projects investigates on-the-fly customization when a change in internal data organization is needed, and while there may be requests in-flight.

Chapter 3

K42 Operating System

K42 is an operating system designed from the ground up and targeted at multiprocessor shared-memory machines. The project is being conducted at IBM Watson Research with collaboration from University of Toronto and other universities. K42 was heavily influenced by Tornado [1, 2, 11, 12, 29], developed at the University of Toronto.

Achieving good performance for shared-memory multiprocessor programs has received considerable attention [5, 12, 15, 19, 29]. K42's overall structure, algorithms, and data structures have been designed with the purpose of achieving good multiprocessor performance, yet without sacrificing uniprocessor performance. K42 uses an object-oriented approach, where every virtual and physical resource in the system is represented by an independent object, ensuring natural locality and independence for all resources. As a matter of principle, all locks are internal to the objects they are protecting, and no global locks are used. Customizability allows us to choose different implementations when we are running an application in uniprocessor versus multiprocessor modes thus maintaining good overall performance. Parallel applications have many specialized needs, from specific memory layout to particular communication and scheduling demands. Here, K42's customizability provides the ability to tailor the operating system to the demands of any specific application.

In addition to the challenges of multiprocessors there are other difficulties faced in operating system design. The requirement of supporting the system across different architectures (PowerPC, MIPS, etc), and having to support a wide range of applications with differing and conflicting resource demands, all contribute to difficulties in achieving good operating system performance. In varying degrees, these challenges cause both programmability and performance problems in operating systems. Programmability issues arise when code for various architectures appear `#ifdef`'ed, making it difficult to understand the code or to modify it. Performance issues arise if code contains conditional code for different hardware platforms, or has to perform in a generic way to meet all possible application demands.

K42 has been designed to alleviate these difficulties by supporting customization in a first-class way. A running K42 operating system is customized for the hardware platform it is running on, and it is customized to present to applications resources that can be tailored to their requirements. This way, K42 achieves the benefits associated with other customizable operating systems that can tune for application behavior. K42's approach to customization also yields significant benefits in terms of structuring and programmability of the operating system and the ability to port it easily and with high performance.

The system, however, is a research operating system in its infancy. We currently have basic Linux libraries running on the system and are working towards being able to run applications compiled for Linux on K42. K42 is expected to be self hosting by fall of 2000, and is available as open source.

3.1 K42 Object Infrastructure

In this section we provide the background on building blocks and clustered objects in order to better understand the dynamic switching mechanisms we describe later. More details of clustered objects can be found in the descriptions of Tornado and clustered objects [1, 12], while a more in depth discussion of building blocks can be found in the HFS paper [16] and an early K42 position paper [2].

3.2 Building-Block Composition

In building-block composition, each virtual resource instance (e.g., a particular file, open file instance, memory region) is implemented by combining a set of objects (e.g., C++ objects) we call building blocks. Each building block implements a particular abstraction or particular policy and might 1) manage some part of the virtual resource, 2) manage some of the physical resources backing the virtual resource, or 3) manage the flow of control through the building blocks. We refer to the overall implementation of a virtual resource as a *building-block composition*.

Customizability is achieved by letting the application specify the particular set (or composition) of building blocks to be used for implementing a virtual resource created on behalf of the application, and by letting the application dynamically change the compositions at any time. This allows, for example, every open file to have a different pre-fetching policy, every memory region to have a different page size, and every process to have a different internal data distribution.

A building block exports an interface that specifies the operations that can be invoked by other building blocks. It may also import (one or more) interfaces that are exported by other building blocks. Two building blocks are said to be *connected* if one of them might invoke operations of the other, and the object is then also said to *reference* the other. Two building blocks may be connected only if the exported interface of the one is imported by the other.

The particular composition of building blocks (i.e., the set of objects and the way they are connected) that implement a virtual resource determines the behavior and performance of the resource. As a simple example, Figure 3.1 shows four building blocks

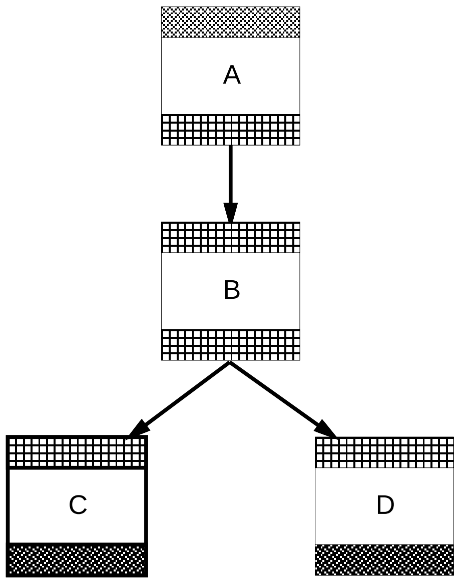

Figure 3.1: *Building blocks implementing a virtual resource such as a file.* C *and* D *may each store data on a single disk,* B *might be a distribution building block that distributes the file data to* C *and* D*, and* A *might be a compression/decompression building block that decompresses data read from* B *and compresses data being written to* B*.*

that might implement some part of a file. B contains references to C and D, and in turn is referenced by A. C and D might each store data on a different disk, B might be a distribution building block that distributes the file data to C and D, and A might be a compression/decompression building block that decompresses data read from B and compresses data being written to B. The imported and exported interfaces are indicated by the pattern at the top and bottom of each object. If two building blocks are connected, then the corresponding imported and exported interfaces must match.

As a concrete example of building blocks, Figure 3.2 shows, in a slightly simplified form, the key building blocks used for memory management in K42. The building blocks depicted in the figure are:

- Process: provides clients with entry points that manage all resources associated with a process,

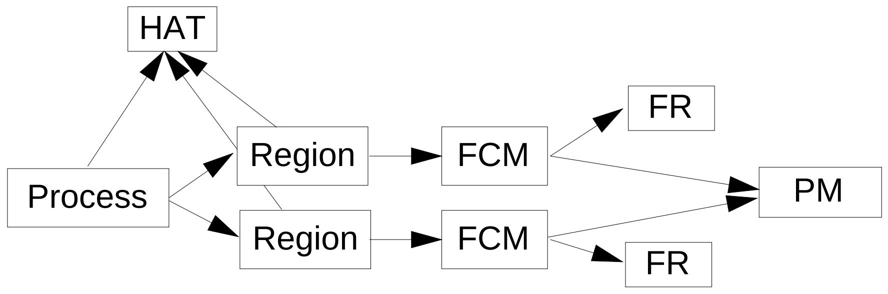

Figure 3.2: *Memory management building blocks in K42. Arrow represents interface import relationship.*

- Region: maintains the mapping of a virtual memory region to the part of the file backing the memory,
- FCM: maintains a memory cache of file pages,
- FR: implements a file object,
- PM: manages physical memory of the system, and
- HAT: performs low-level hardware address translation.

The arrows in the figure represent the interface import relationship. For instance, the Region building block imports the interface exported by the FCM building block.

As an example of these building blocks in action, in a page-fault, the exception is delivered to the *Process* building block for the thread that faulted. The Process maintains the list of mapped memory regions in the process's address space, which it searches to identify the responsible *Region* to forward the request to. The Region translates the fault address into a file offset, and forwards the request to the *File Cache Manager* (FCM) for the file backing that portion of the address space. The FCM checks if the file data is currently cached in memory. If it is, then the address of the corresponding physical page frame is returned to the Region, which makes a call to the *Hardware Address Translation* (HAT) to map the page, and then returns. Otherwise, the FCM requests a new physical page frame from the physical memory manager (PM) and then asks the *File Representative* (FR) to fill the page from a file. The FR then makes a call to the corresponding file server to read in the file block. The thread is re-started when the file server returns with the required data.

In this example, the composition of these building blocks together manage two virtual memory regions backed by different files. Since there are multiple implementations of each of these building blocks (e.g., there are uniprocessor- and multiprocessor-optimized building block implementations for the Region interface, and there are FCM's optimized for small files and big files), together they form a highly customizable collection of virtual-memory-mapped file implementations for different resource usage patterns.

It is important to note that *each* virtual resource instance is composed of different building-block instances. For example, in Figure 3.2, each region and each FCM has a different object instance implementing it and therefore each could be a different implementation. One region and FCM could be optimized for small file and uniprocessor accesses, while the other pair may be optimized for large file and multiprocessor accesses. Here, customizability is achieved by choosing different building blocks.

Further customization can be achieved by modifying the topology of the composition. The topology defines in the abstract which type of building block connects to which other type of building block. As an example of modifying the topology, in K42 we have a building block specialized for copy-on-write (COW) behavior. In Figure 3.2 it would be added between the region and the FCM. To do so, the COW accepts as input what the region outputs, and the COW outputs what a standard FCM accepts as input. Building blocks can also present a different interface to applications. For example, in creating a region, one could have an additional parameter stating a maximum size.

After an application instantiates a building block to manage a resource and before using it, the system verifies the building-block composition for type safety to ensure the correct interfaces have been implemented (i.e., the corresponding imported and exported interfaces match).

K42's building-block composition separates mechanisms for customizability and extensibility. Informally, a system is extended when new functionality (i.e., new code) is added, and customized when an application specifies the functionality to be invoked on its behalf. As described in Chapter 2, previous approaches have achieved customizability through extensibility, i.e., customized code is loaded into the system. In K42, customizability is achieved by allowing applications to compose building blocks from a set of existing components to manage a resource. There is no performance degradation due to downloading or verification of the components. Also, since each application independently chooses a building-block composition appropriate for its requests to a resource, the system provides customizability without affecting other applications building-block composition for their view of the resource. For example, one application can use a shared representation of a process object while another application can use a distributed representation. Extensibility is achieved by allowing trusted parties to write new objects that extend the functionality of the system. The separation leads to lower overhead. And as with other composable object-oriented systems, K42 has the benefit of better maintainability and improved multiprocessor performance.

The goal of customizability is to match the implementation of a given building-block composition to the needs of the application using it. A static choice of building blocks can be sub-optimal for several reasons. The operating system does not generally know *a priori* the expected usage pattern of a resource by an application. This is true for most mixes of applications run by an operating system. The system therefore can only guess the object best suited for an application. Even in the case where an application provides hints on how it will use a resource, its requirements may change over time. These reasons motivate the ability to dynamically change building blocks when they are in use.

Building-block compositions provide a powerful method for customization, useful for multiple purposes. The ability to change a building block on the fly adds to its usefulness.

In the next section we describe the clustered object infrastructure needed to support on-the-fly customization.

3.3 Clustered Objects

Optimizing performance across the multiple and potentially competing operating system services is complex. To reduce the complexity of this problem, K42 takes the building-block approach (as described in the previous section) to system design by breaking the management of resources into logical pieces or objects. However, this alone does not necessarily achieve good multiprocessor performance, since some applications stress concurrency when going for high throughput. Also, locality of reference needs to be maximized to avoid slower remote memory accesses and reduce cache-coherence traffic. Clustered objects extend the object-oriented design of building blocks by providing the additional ability of managing the level of distribution of data and locality of execution. Based on Tornado, K42's clustered object infrastructure provides a framework for controlling concurrency and locality of reference in objects [1, 12, 29].

3.3.1 Overview

From a client's perspective, clustered objects appear similar to C++ objects, i.e., their interfaces are the same. A clustered object is logically a single object, but internally it is composed of one or more component objects called *representatives*, or *reps*. Each rep handles calls from a specified subset of the processors (see Figure 3.3). A clustered object is accessed via a *clustered object identifier*. The method invocations on clustered objects are done using this identifier. Each call is automatically directed to the appropriate rep based on the processor from which the call was made and on the *degree of clustering*. The degree of clustering determines how many reps there are in the system. There might be one rep for the entire system, one rep per processor, or any other appropriate mapping

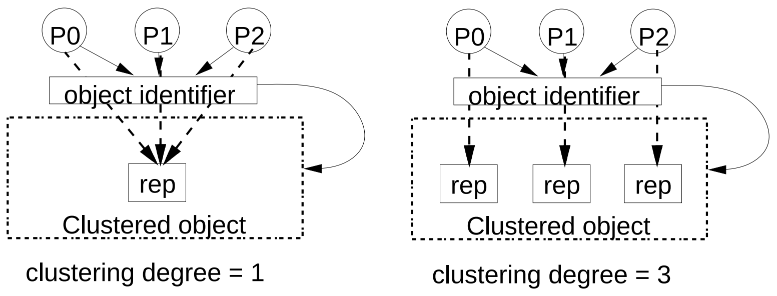

Figure 3.3: *An abstract depiction of a clustered object.*

such as one rep per NUMA node in a NUMA system. Each representative provides its clients with the full functionality of the clustered object, and if necessary, the reps of the clustered object communicate with each other to maintain consistency.

The internal data representation and algorithms of the clustered object is transparent to the client. If the shared object data is read-mostly, replication may be adopted, with each processor's local rep maintaining its own replicated copy of the data. Some objects are best partitioned so that the data most accessed by a processor will stay in the rep local to that processor. With appropriate internal implementation, an object can be optimized for locality and concurrency depending on an assumed access pattern. With an implementation involving multiple reps, it is necessary to keep them consistent. While the internal implementation and data distribution of a clustered object can be modified and fine-tuned to suit its locality requirement, the interface that it exposes to its clients remains the same. While the internal data may be replicated, migrated, or partitioned, the clients can make method invocations to the object without knowledge of its actual implementation.

The clustered object infrastructure in K42 has a number of benefits. It provides a framework to optimize objects for locality and concurrency using commonly applied techniques such as replication or partitioning. These techniques can be applied both to data structures and locks. The interface exposed by the clustered object isolates the internal organization of the reps from the clients. Also, clustered objects facilitate incremental optimization and experimentation for each system object. A system object can be implemented initially as a single-rep clustered object, whose implementation would be almost identical to that of a common non-clustered object. If the object becomes a bottleneck, a multi-rep clustered object could be implemented and used instead. Since the interface remains constant, implementations with different degrees of clustering and consistency protocols can be experimented with, without modifying the rest of the system. The interface provides the flexibility to allow different implementations of a clustered object to exist, optimized for different usage requirements. As discussed in Section 3.2, in K42, this flexibility of allowing different customized implementations of the same object is referred to as building-block composition.

3.3.2 Implementation

In this section, we present the implementation of the clustered object system in K42. We describe how clustered objects are referenced and what system-level data structures are used. We then explain how reps are created within a clustered object.

Object translation table

Clients access a clustered object by means of a *clustered object identifier*. In the K42 implementation, the identifier is a pointer to an entry in a per-processor table called the *object translation table* (OTT) (see Figure 3.4). The entry in the OTT points to the rep associated with the clustered object (once it has been used). Because the OTT is defined on a per-processor basis, for each clustered object identifier, there is one object translation entry per processor. The entry on each processor for a particular object could point to the same rep, or to different reps, depending on the degree of clustering. Using the extra level of indirection introduced by the object translation table, the distribution of internal object data can be optimized independently of the interface.

To allow clustered object invocations on each processor with the same identifier (pointer), we exploit K42's aliased virtual memory capability, which allows the same virtual memory address to be mapped to different physical addresses on different processors. Per-processor aliased virtual memory regions are used within the address space to give each processor its own unique copy of the object translation table, located at the same virtual address. Since many objects are only accessed on the processor on which they are created, we partition the ownerships of the table into disjoint subranges, one per processor. This way, allocation of entries in the table does not require synchronization across processors.

Miss handling

Representatives are lazily created. This is done for a couple of reasons. Requests to a particular clustered object are not necessarily made from all processors; for some clustered objects only a small subset of processors make requests. Further, lazy creation spreads out the creation time to first use.

Lazy creation is accomplished by initially installing a reference to a generic object handler (instead of to a rep) in all the object translation entries. An object table entry is modified to point to a particular rep on demand when the processor's first method invocation on the clustered object is made. This way, processors that do not access a particular object will not need to perform unnecessary set up. The process of setting the translation entry to point to a rep is called *miss-handling*. The processor that incurs the miss is said to be *faulting*.

Different clustered object implementations may have different ways of handling misses. In particular, in a multi-rep clustered object with one rep per processor, we manage the set of reps, and create a new rep if the rep corresponding to the faulting processor is not already in the set. In our clustered object system, the object that manages the set of reps is called the *root object*, or just *root*. The root is responsible for object-specific miss-handling, and is instantiated when the clustered object is instantiated. The pointer

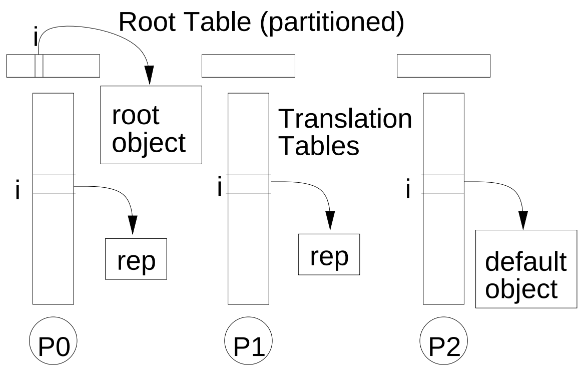

Figure 3.4: *Clustered object implementation. Clustered object i has been accessed on P0 and P1, where reps have been installed; P2 has not yet accessed object i.*

to the root is installed in an auxiliary table called the *root table*, indexed the same way as the object translation table. Besides being responsible for rep creation and initialization, the root is also responsible for maintaining shared resources used by all the reps in a clustered object.

Entries in the object translation table are initialized to point to a generic handler object called the *default object*. The default object redirects calls to the corresponding root object and invokes its object-specific miss-handling code. The result of invoking this miss-handling code is a pointer to the rep responsible for handling the call. The default object forwards the original invocation to the rep, and this forwarding is transparent to both the rep and the client that faulted. If the root installed a pointer to a rep in the object translation table during miss-handling, then subsequent method invocations to the clustered object will be handled by the rep directly (see Figure 3.4). This mechanism allows clients to invoke methods of the clustered object, without knowing that a miss-handling operation may take place.

The default object leverages the C++ virtual function mechanism to perform miss-handling transparently. We allocate and assign to the default object a virtual function table with enough generic virtual functions to support expected objects. However, if necessary, the size of the vtable can grow dynamically.[1] Since the default object knows nothing about the invocation context, each virtual function of the default object saves all the registers of the original caller before performing the miss-handling work. Once the miss-handling is done and a rep is obtained, it restores all the register content, replaces the `this` pointer argument with the pointer to the rep that handles the call, and forwards the call to the corresponding method of the rep by looking up the rep's virtual function table. While the operation has non-negligible overhead, it is performed only once for establishing the translation entry and infrequently thereafter.

A consequence of using the vtable for miss-handling and call redirection is that all clustered object methods will need to be virtual. This is acceptable since the object polymorphism is used in many system designs anyway.

Paging support

The translation tables are likely to be sparsely populated, because there can be a large number of clustered objects, and because clustered object identifiers are allocated from the subrange assigned to the processor they are created on. As a result, we choose to make the translation table memory pageable. Further, since the translation tables are sparse and represent a cache (i.e., entries can be regenerated by the root objects which maintain the reps at any time), a victim page can just be discarded rather than paged out. Future accesses to the clustered objects in this range will cause the default object to re-handle the misses, which restores the entries. If necessary, a dense compression table can be used as a second-level cache of translation entries to reduce the extra miss-handling caused by discarded pages.

[1]A special virtual memory region can be used to back this table. When a page fault on this memory region occurs, a memory page is supplied with the new virtual function pointers, which point to newly initialized generic functions. Currently, K42 provides a vtable that has many more entries than exist in the system's largest clustered object.

Cross-address-space invocation

Cross-address-space object method invocation is supported by instantiating a local proxy object, which communicates with the remote interface object associated with the server object. Parameter marshaling and caller authentication are performed by this local proxy object.

3.3.3 Generation Count

Another portion of the infrastructure used by the dynamic object switching infrastructure is K42's *generation count*. This is used by the clustered object facility for garbage collection [12]. The facility for remote procedure call in K42 is called *Protected Procedure Call* (PPC). In the PPC model, a request to a server (including the kernel) gets executed by a (logically) new thread that is created on the server side for handling a PPC request, and ends when the request is satisfied. In K42, all requests to server objects made by external clients are accomplished via the PPC facility.

A *reference count* tracks the number of active requests (threads) executing in an address space on a per processor basis. When a thread is created, it is assigned a *thread generation*. An epoch is the time period that starts when the first thread in a generation is created and ends when there are no longer any threads that were started in previous epochs. The generation count (or epoch number) is advanced when the epoch is over, and this count is used in thread generation assignment. To efficiently determine if the generation count can be advanced, we need at least two reference counts, one that counts the number of threads with the current generation and one with the previous generation. If the system only maintains the reference count for the current generation, then the generation count may not be advanced. This is because the reference count may never reach zero, and so the system cannot determine that the epoch is over.

K42 maintains reference counts for two generations — the current generation and

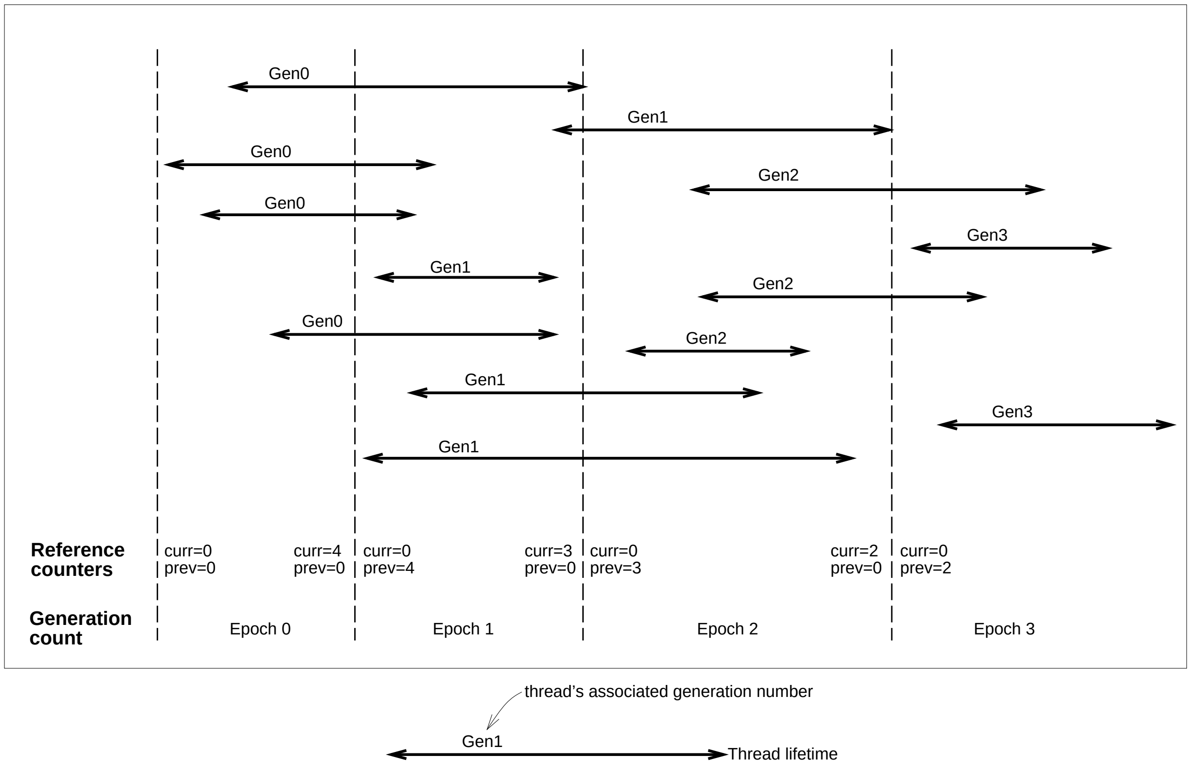

Figure 3.5: *Thread lifetime diagram illustrating the generation-count updates.*

the previous generation. When a new thread is created, the system increments the reference count for the current generation, and the thread is assigned with that thread generation. When the thread terminates, the corresponding generation's reference count is then decremented. This way, the generation count is guaranteed to advance since once the current generation is started, the previous generation's reference count will only decrease. The generation count can advance when the previous generation's reference count reaches zero. Under this model, we can determine that there are no more running threads started prior to any particular time, by recording the current generation count at that time and waiting for the generation count to increase by at least two, since this implies that all the threads from both the current and the previous generations have terminated.

Figure 3.5 illustrates how the generation count relates to the reference counters as described above. When the previous generation's reference count (*prev* in the figure) is zero then the generation count can be advanced, and the reference counts are updated accordingly.

3.4 Summary

Building blocks allow flexible customization of operating system resources, while clustered objects provide system-level support for building blocks optimized for concurrency and locality in multiprocessors. Building blocks provide for better programmability and maintainability as well as allow for easier porting with better performance, compared to the other more radical approaches of customizability through extensibility (described in Chapter 2).

Since K42 is designed for multiprocessors, most building blocks are implemented as clustered objects. A clustered object is one whose implementation is potentially distributed across a multiprocessor for concurrency and locality. A clustered object is logically (i.e., externally viewed as) a single object, but it is internally composed of one or more component objects called representatives. Each representative handles calls from a specified subset of the processors.

The clustered object model is a partitioned object model which allows for expressing locality and concurrency optimizations in a consistent manner. It is designed to provide the benefits of an object-oriented paradigm, such as clear separation between interface and implementation, better maintainability with modularized code, and improved programmability via the inheritance hierarchy. The clustered object translation table provides the flexibility necessary for clients to access the object through a local representative object transparently, and thus allows the degree of data distribution to be changed without affecting the interface.

While it is impossible to completely remove interactions between processors and eliminate remote memory access, the clustered object mechanism allows fine-tuning at the object level so that such interactions are done only when necessary, thus achieving maximum concurrency and locality of reference.

Clustered objects allow us to explore and implement locality and concurrency optimizations while presenting the same interface to applications. The building-block architecture provides a practical framework for system design; most kernel objects and system servers in the K42 operating system are built using building blocks implemented by clustered objects. In the next chapter, we describe how these building blocks can be exchanged for new ones with the same interface on the fly at run time, transparent to the clients that are using them.

Chapter 4

Dynamic Object Switching

As described in Section 3.2, building blocks allow custom composition of system objects, providing the flexibility to tune performance on a per-application and per-use basis. In K42, an application programmer can request system resources from building blocks that are customized to the needs of that application. However, if an application's resource request pattern changes over time — for example, when the application enters another execution phase — then the originally chosen object may no longer be optimal. Similarly, in many cases, the programmer does not know, or cannot specify *a priori* how an object will be used, thus requiring the operating system to use a default choice or attempt to infer how the object will be used. In this common scenario, the operating system can improve application performance if it chooses an initial building block for the object, and then dynamically switches to another implementation of that building block when the application's request pattern to the resource provided by the object changes. The implementation switch may be initiated by a system performance facility which monitors the request patterns to the object, or by the program using the object if the change in usage pattern is known at compile time.

The ability to perform dynamic, post-creation switching of system objects can be an important aspect to achieving good performance in customizable operating systems, as we will show in Chapter 5. Dynamic switching complements the flexibility offered by the static, creation-time customization via building blocks. It allows appropriate customization across a wider range of applications by being adaptive to changing requirements of applications. In this chapter, we describe the design and implementation of the K42 dynamic clustered object switching facility.

4.1 Background

There are different reasons why the operating system may want to dynamically switch an object. In increasing degrees of complexity, it may wish to switch to an object that provides a different policy or algorithm while using the same internal data structures, or switch to an object with a different implementation, having a different internal data representation.

Most often, a change in policy implies a change in the associated internal data structures. This is common in K42, since the clustered object system abstracts away the internal data distribution, providing an additional degree of flexibility frequently taken advantage of by programmers, and hence adding a dimension in which the internal data organization can be tuned to the access behavior. When a change in the underlying data structures is needed during a switch, the switching operation can become quite involved, especially if the object is highly concurrent and event-driven. Multiple threads are likely to be servicing requests to the object at any instance. We call requests that are being serviced by threads executing in the object when a switch is initiated *in-flight* requests. In-flight requests complicate switching because unless the change in data can be propagated to the new object coherently, the switch cannot occur until all the in-flight requests are serviced by the original object.

When the internal data representation remains unchanged, there are obvious optimizations to the dynamic switching mechanism to simplify the problem. As an example of such an optimization, consider switching between two objects implementing different page replacement policies, when the new object changes the victim page selection policy while continuing to use the same internal data structure. As long as the new implementation coherently accesses the *same* internal data members as the old one, we can have concurrent requests being serviced by both the old and the new objects. For these situations, no explicit data coherency needs to be maintained, and the internal data being used are not duplicated or relocated. Therefore, the new object implementing the new policy can replace the old one without concern for active requests being serviced in the object. In such cases, it is sufficient to direct new requests to the new object, which is accomplished by pointer manipulation in K42's clustered object translation tables. This is easily accomplished in K42 due to the extra level of indirection introduced by the object table. Without the object translation table, even this level of dynamic switching could prove challenging.

This dissertation focuses on solving the common and complex situation where the interface exported by the object is the only thing guaranteed to remain unchanged.

4.1.1 Hybrid object

There are a couple approaches to dynamic switching one might take. The simplest would be to design a *custom hybrid object* (see Figure 4.1). A custom hybrid object is a self-switchable object containing two or more implementations internally, all coordinated by the switching wrapper, which implements the switching logic as well as the object interface to external clients. There are deficiencies with this model, however. To understand them, we present a brief overview of what this approach might entail, and then describe its disadvantages before presenting the approach we took in K42. As described in the previous section, we cannot perform a switch if coherent data transfer is needed until all the in-flight requests are serviced by the original object. Without external system support, a custom hybrid object would need to maintain a count, tracking the number

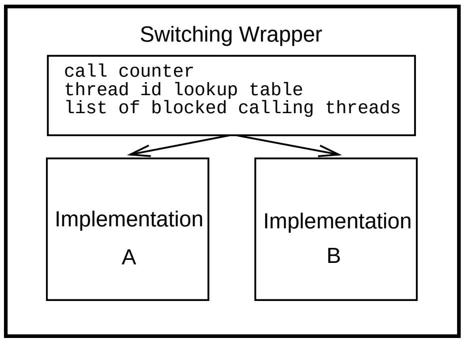

Figure 4.1: *A custom hybrid object with multiple implementations.*

of requests the object is servicing at all times. After switch initiation, it would have to block new requests while in-flight calls are allowed to finish processing. The reason for blocking new calls is to ensure that the call count will eventually drop to zero. When all in-flight calls have returned (i.e., the call count reaches zero), the state of the original object can be safely transfered to the new one. After the state transfer, all the blocked calls can now be resumed and forwarded to call the new implementation. The complexity increases when the object is re-entrant. Blocking a re-entrant call causes deadlock since the in-flight call count would then never reach zero. To solve this problem, the switching wrapper would have to remember the threads that are currently in-flight. When a new call enters the wrapper it checks whether the calling thread is already in-flight. If so, the call to the object is re-entrant and hence should not be blocked.

The main disadvantage of this approach is the added overhead to the normal call path, even when the object is not trying to switch implementations. This overhead includes keeping track of the in-flight call counter and thread identifiers *at all times*. Also, the switching wrapper has to be custom built to track all calls into the object and forward calls to the corresponding method of the appropriate object. This wrapping object, when tightly coupled with the objects to be switched between, can significantly increase the complexity (and decrease the maintainability) of the original objects. Another disadvantage is that this would have to be custom implemented for each object desiring the capability to switch. Because of these disadvantages, we have designed a more generic facility that incurs no overhead when not switching objects.

4.1.2 K42's dynamic switching design goals

K42's approach to dynamically switching objects has the following design goals:

- zero impact on performance when an object is not switching,
- minimal code impact on the objects to enable switching between different existing object implementations,
- zero impact to other system objects unrelated to the switching operation,
- good performance and scalability; that is, the switching operation itself should incur low overhead and scale well on multiprocessor systems, and
- switch transparency; that is, clients using the building block being switched need not be aware that the implementation behind the interface is being switched.

4.2 K42's approach to dynamic switching

K42 takes an approach similar in concept to the hybrid object, but it adds no cost to the object when not switching and the implementation is generically used by all objects wishing to support dynamic switching. In addition, K42's dynamic clustered object switching facility provides a common switching interface and implementation that can be used by any clustered object desiring to perform a dynamic switch.

We use two aspects of the clustered object infrastructure for dynamic switching that were described in Section 3.3: the Object Translation Table (OTT) and the generation count. The OTT provides a level of indirection that allows us to intercept method invocations. The generation count allows us to track in-flight requests, helping us determine when all requests made prior to switch initiation have finished. To start a switch in K42, the object translation table for the switching object is modified to point to an interposing clustered object called the *mediator* (see Figure 4.2). This mediator object is a generic object capable of handling the switching of any clustered object. This object mediates calls from the time the switch has been initiated, to when the switch has completed. It intercepts clustered object method invocations made to the original object and transparently counts and tracks new requests. When it determines that all in-flight requests are accounted for (so all the re-entrant calls can be identified), it blocks all new incoming calls and waits until the tracked in-flight calls have completed. Once this condition is met, i.e., there are no more in-flight calls using the original object, the mediator initiates the data transfer between the old and the new objects via a callback and then redirects the blocked calls to the new object.

Compared to the hybrid object implementation described above, this approach separates the complexity of switch-time in-flight call tracking and deadlock avoidance from the implementation of the object itself. Call interception and mediation are simplified by the clustered object system infrastructure. Besides the data transfer callback, the rest of the switching process is automated by the K42 mechanism, allowing for easy addition of objects that wish to take advantage of dynamic switching into the system. Details of the implementation are presented in the next section.

4.3 Implementation overview

Figure 4.2 illustrates the states of the objects involved in switching between the clustered object identified by i from implementations A to B. Initially, implementation A is used when clustered object identifier i is called (Figure 4.2a). It is assumed that both objects A and B are already instantiated. When the switch is initiated, a *mediator* object is created, and a reference to it is installed in the OTT in slot i. Subsequent object invocations

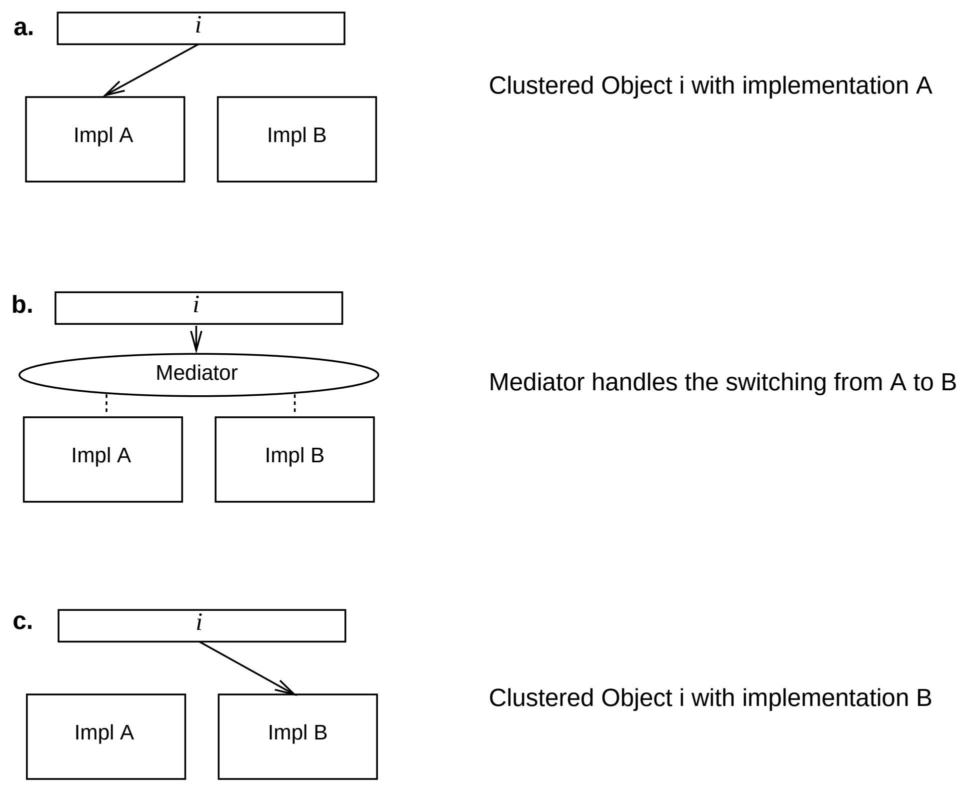
a.
i
Impl A
Impl B
Clustered Object i with implementation A
b.
i
Mediator
Impl A
Impl B
Mediator handles the switching from A to B
c.
i
Impl A
Impl B
Clustered Object i with implementation B

Figure 4.2: *A switch from implementations A to B of clustered object i.*

for i then invoke the mediator object instead of the original object (Figure 4.2b). The mediator object has references to both implementations, and it is responsible for tracking new incoming clustered object method invocations to i. The mediator, depending on the state of the switching operation, will either forward the call immediately to A, block the thread associated with the incoming call, or forward the call to implementation B. There are three phases associated with the switching operation: *Forward*, *Block*, and *Completed*. They are described in the following paragraphs. When we reach the *Completed* switch phase, the mediator modifies the OTT to remove itself from the i-th slot and have the slot refer to B instead (Figure 4.2c) so that all future calls to i are handled by B directly.

During the *Forward* phase, the mediator tracks new incoming calls by their thread identifiers and increments an in-flight call counter. It decrements the counter when these invocations return. The mediator stores the thread identifiers in a hash table so that re-entrant clustered object method invocations by the same thread can be identified and allowed to continue even during the *Block* phase. This is required to prevent deadlock, which would occur if we blocked a re-entrant thread; i.e., since the thread has been previously counted, if it is not given a chance to finish, the in-flight counter would never reach zero, and we would never make forward progress. The hash table is also used to save register values used for transparent call forwarding and call returning. The *Forward* phase continues until we have gained knowledge of all in-flight calls to the object; that is, there are no more in-flight requests that were started prior to the switch initiation. We know there are no more such requests when the generation count described in Section 3.3 has advanced. If we start blocking incoming threads too soon without being aware of all requests, we may cause deadlock because a thread we did not know about may have been re-entrant.

The *Block* phase starts when the mediator determines there are no more in-flight calls that were started prior to the switch initiation, i.e., all in-flight calls are accounted for in the hash table established during the *Forward* phase. The mediator determines this when the generation count advances, guaranteeing all requests made prior to switch initiation have finished. During the *Block* phase, new incoming calls are first checked to see if they belong to one of the in-flight threads tracked by the hash table. If so, it is a re-entrant clustered object method invocation and is forwarded to the original implementation A. Otherwise, the thread is a new incoming thread, which the mediator blocks until the switch has completed, at which point it is unblocked and forwarded to the new implementation B. There are more complex issues related to indirect recursion, but they can be adequately handled.[1] Blocking new invocations will stop the in-flight call count from increasing. Once the call count reaches zero, there are no more threads executing within the object A and the mediator initiates a data transfer, which transfers the state of the original object to the new object so that subsequent requests to the new object are serviced coherently to the state of the original object. While this phase sounds

[1]Details of the such issues will be described in Section 4.4.2.

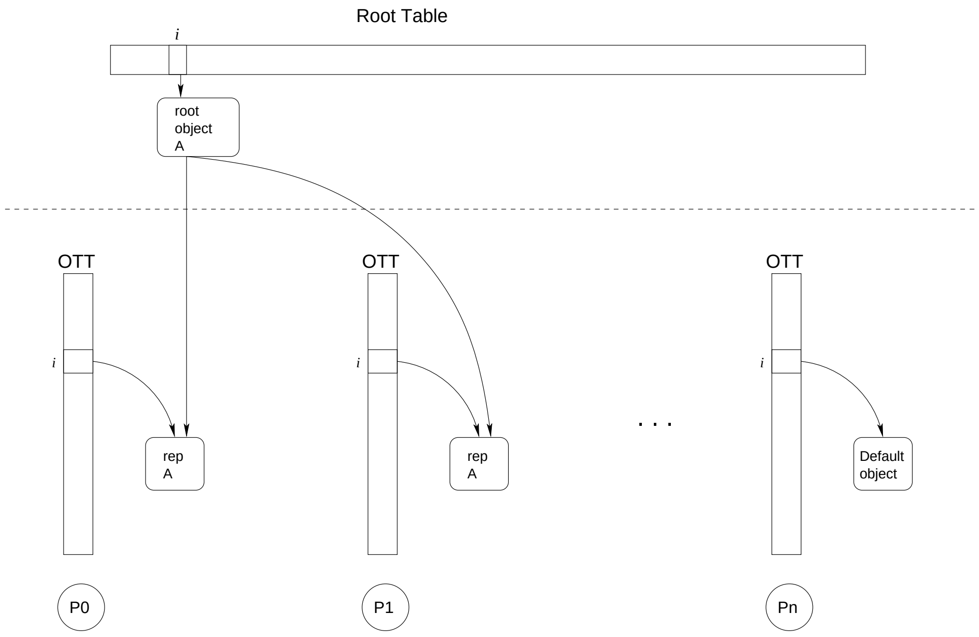

Figure 4.3: *State of the clustered object space before the switch. Clustered object i is using implementation A and reps are installed in the OTT's of P0 and P1. Pn has not yet accessed clustered object i.*

complicated, it is reasonably straight-forward and occurs very quickly after the forward phase in practice.

In the final phase, called the *Completed* phase, the mediator removes its interception to the clustered object i and installs implementation B to handle future calls. All the threads that were blocked during the *Block* phase are unblocked and these calls are forwarded to implementation B. Calls that enter the mediator object after the *Block* phase is completed, but before the pointers have been changed to point to the new object are forwarded to the new object implementation B.

To better understand how the clustered object system is leveraged by the switching mechanism, Figures 4.3–4.6 illustrate the states of the clustered object i of the process during a switch. Figure 4.3 shows the state of the system before switch initiation. Processors $P0$ and $P1$ have reps of implementation A installed in the OTT entries. When the object switch to implementation B is initiated, the mediator root is installed in the

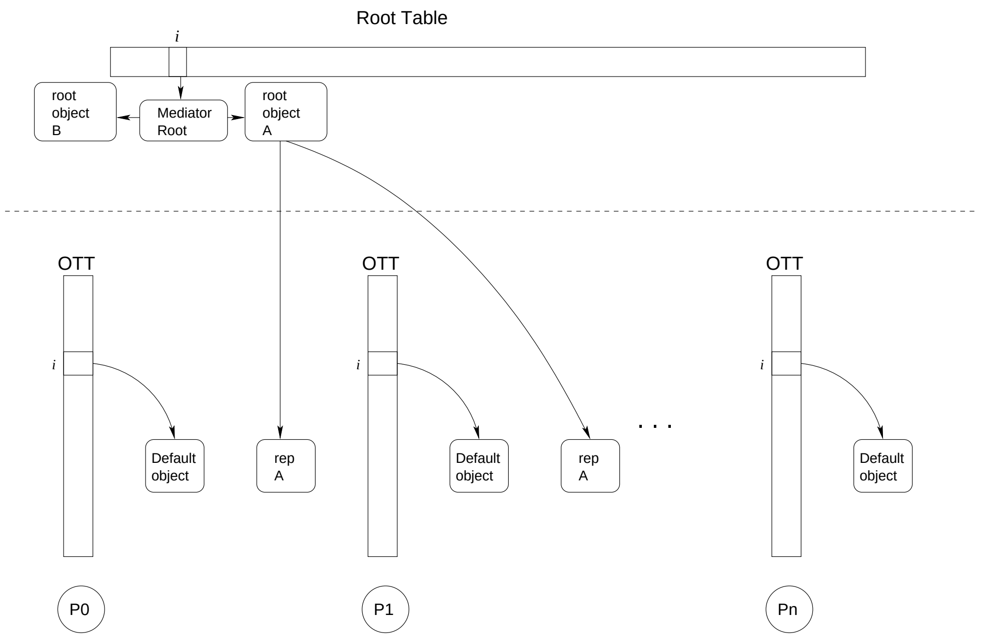

Figure 4.4: *Initiated a switching to implementation B for clustered object i. OTT entries are flushed and the mediator root has taken over the root table entry for i.*

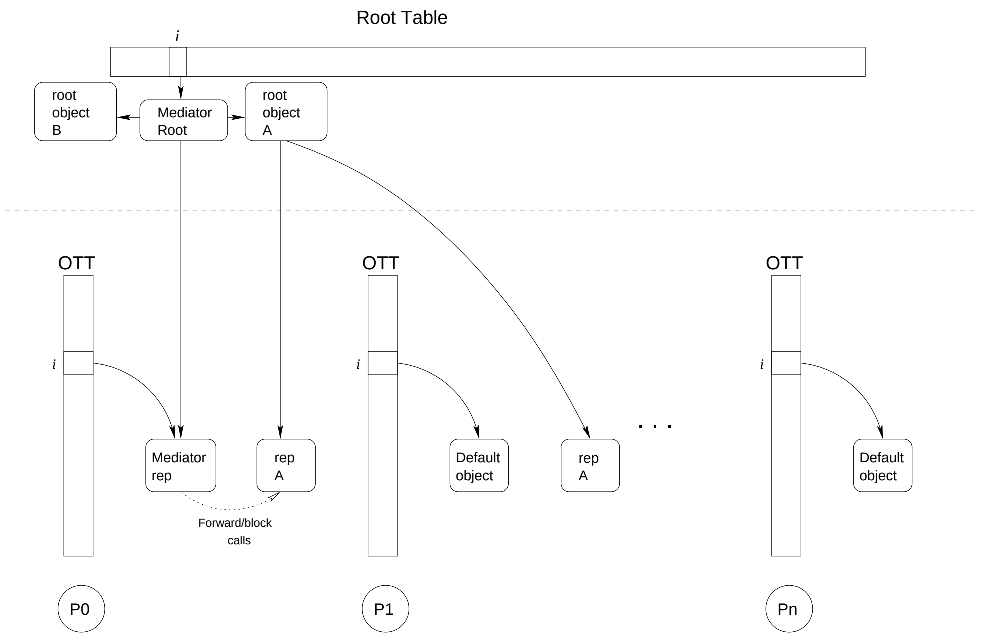

Figure 4.5: *Call mediation is in progress. On clustered object access, OTT entries are set to point to the mediator reps which perform call mediation. P0 has a mediator rep installed.*

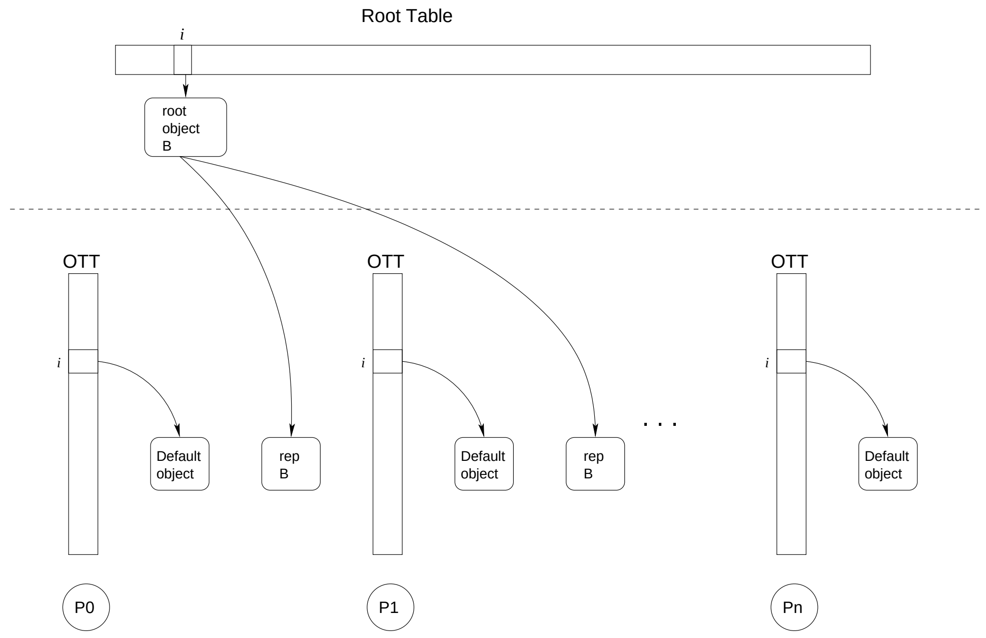

Figure 4.6: *Switching to implementation B completed. OTT entries are flushed and will set to point to the reps of B when clustered object i is invoked.*

root table and the OTT entries for slot i for processors $P0$ and $P1$ are “flushed” in that they are made to refer to the default object. The subsequent state is shown in Figure 4.4. When new invocations are made to the clustered object, the invocation faults (since the i-th slot in the OTT refers to the default object) and a mediator rep is installed using the standard mechanism described in Section 3.3. The mediator reps then mediate all accesses to implementation A from processor $P0$ as described. Figure 4.5 shows the state after processor $P0$ made an invocation on clustered object i after switch initiation. At this time, slot i in $P0$'s OTT refers to the mediator rep, which was installed by the mediator root as part of the standard miss-handling procedure. When the *Completed* phase started, the data transfer is done from A to B and root B is installed in the root table. The OTT entries are flushed again so that new accesses will be handled by the root and reps of B directly. The state is shown in Figure 4.6.

Figure 4.7 illustrates the different phases of switching within the timeline of a process, along with the lifetimes of the threads that could be executing within the process. At

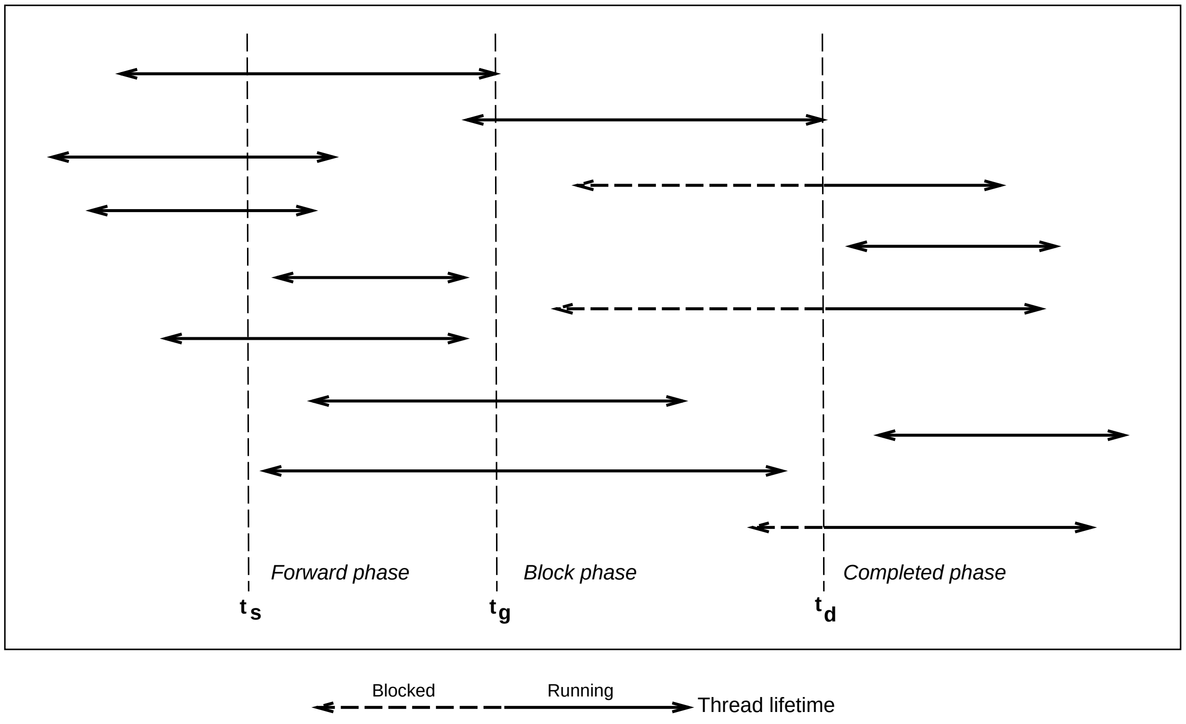

Figure 4.7: *Timeline showing different phases and their association with the lifetimes of the threads in the process.*

time t_s, a dynamic switch is initiated, and the mediator takes control over the clustered object to be switched. It marks the start of a thread generation *epoch*, and the *Forward* switch phase starts. Threads started prior to t_s are not tracked by the mediator, and the generation count mechanism is used to ensure that these threads have terminated before the *Block* phase commences. At time t_g, when a new epoch starts, the mediation enters the *Block* phase. Threads started prior to the *Block* phase (but after t_s) will continue running until completion. Note that these threads are created during the *Forward* phase and are therefore fully tracked by the mediator. New threads that are started during the *Block* phase (i.e., after time t_g) are blocked. This way, the *Block* phase will finish when all the threads started during the *Forward* phase have completed execution. This is determined by the mediator's in-flight call counter. At time t_d, the last in-flight call has terminated and so there are no more threads executing within the object to be switched.

At this point the data transfer is started to transfer the object's relevant state to the new object, and the blocked threads are then allowed to run, using the new object. Subsequent threads invoking the clustered object will be serviced by the new implementation.

K42's generic switching mechanism allows any object to be switched with any other object that implements the same interface. The generic mechanism is more difficult to implement than the custom object, because the generic mechanism has no association with the object it is switching; hence: 1) it cannot store anything on the stack because it has to invisibly interpose itself between the caller and callee, 2) it does not have an obvious place to track transition data associated with the specific object it is switching, i.e., an in-flight count and a thread id hash table, 3) it needs to transparently intercept the return call to decrement the reference counter and delete the thread id from hash table, and 4) it needs to keep track of who to return to after the request is completed. How this is achieved is described in the next section.

4.4 Implementation details

In this section, we provide some details of the K42 implementation of the dynamic clustered object switching mechanism. We describe the implementation of the mediator clustered object such as when the mediator is installed, how it intercepts new calls, and how it determines when there are no more in-flight calls to the original object. We also discuss multiprocessor issues regarding the mediator design. Deadlock avoidance is a major part of the dynamic switching solution and is discussed in detail here. Issues and options regarding data transfer are also covered.

4.4.1 The Mediator

Clustered object method invocations are made through the clustered object identifier i, a pointer in the clustered object translation table (OTT). Upon switch initiation we instantiate the mediator clustered object. For the duration of the switch, the mediator intercepts calls to i. This is accomplished by swinging the pointer in the root table to point to the root of the mediator and performing a *flush*. The flush resets the pointers in the object translation table that were pointing to the reps of the clustered object to point back to the default object (as described in Section 3.3). From that point on, new invocations result in object table translation misses and are handled by the root of the mediator in a standard way. As a result, mediator reps are in control, performing call mediation for the original clustered object.

At switch initiation, worker threads are created on the processors whose object translation entries are pointing at the reps of the running clustered object. The worker threads perform the flush of those entries. These threads are needed to perform the flushes since the translation entries are located on processor-specific regions of memory and can only be accessed locally. These threads are also responsible for performing thread generation checks to determine when there are no more in-flight requests that started prior to mediation. Currently this is accomplished by polling the generation count. We plan to examine the benefits of being notified proactively (i.e., the system invokes some notification callback registered by the switching layer when the generation has elapsed), although this would require an additional check on every thread completion.

The mediator object is designed to handle call mediation for any clustered object interface transparently. The generic call mediation code consists of the mediator rep vtable, the common mediation routine, the mediation prolog, and the mediation epilog. The mediator representative's virtual function table contains pointers to methods that act as trampolines[2] to the common mediation routine (see Figure 4.8). The prolog and epilog of the mediator perform pre- and post-processing associated with call mediation, respectively. The main purpose of the prolog is to examine the current switch phase

[2]These methods simply record the virtual function table index and jump to a common routine. Such simple methods are commonly referred to as trampolines in operating systems.

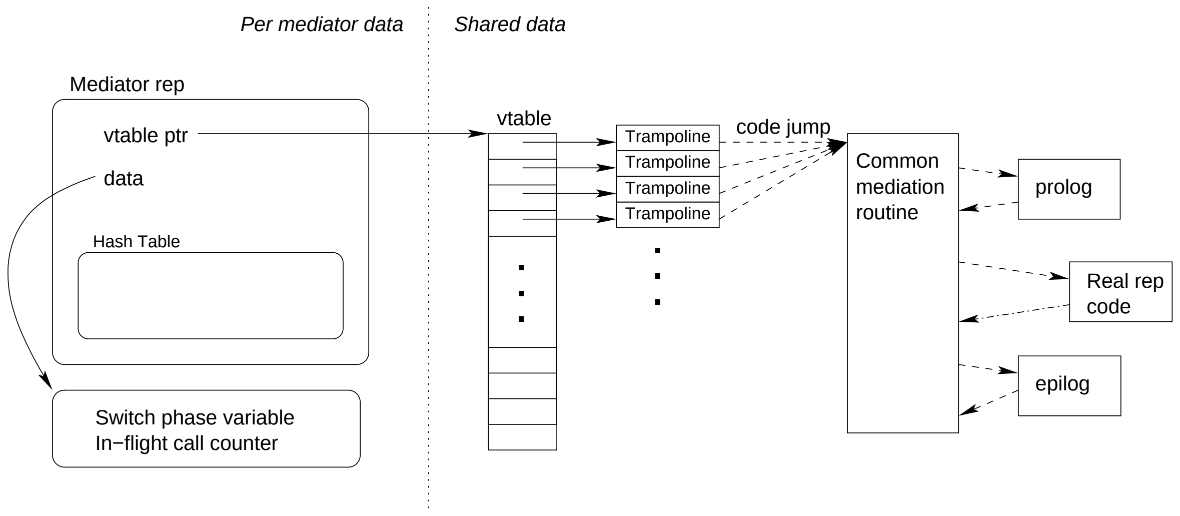

Figure 4.8: *Implementation of the mediator representative object.*

and decide what to do with each calling thread: i.e., *i*) forward it to the original object, increment the in-flight call counter and record the thread identifier in a hash table, *ii*) block it until the switch is complete, or *iii*) directly forward it to the new object. The prolog is called by the common mediation routine, before forwarding the call to the actual method of the target representative. The epilog code is called if the forwarded call returns to the common mediation routine. It is invoked only when the prolog has forwarded the call to the original object. The epilog decrements the in-flight call counter and removes the thread identifier from the hash table, The common mediation routine acts as the assembly glue to save and restore registers, invoke the prolog, the actual method of the forward target, and the epilog when needed.

This mechanism of forwarding calls generically is similar to that of the default object call redirection described in Section 3.3. However, our call mediation has the added complexity of needing to catch the call returns on the way out to first run the epilog code. Since the prolog determines whether epilog processing is necessary, it returns a flag to the common mediation routine so that the mediation routine can determine whether it should simply *jump* to the target method (i.e., the call will not return to the mediator) or it should *call* the target method (i.e., *jump-and-link*, the call will return to the mediator for epilog processing).

To allow the epilog to be called, the prolog routine has to save the return address of the original caller so that it can return back to the original caller after the call returns to the mediator for epilog processing. During a normal method invocation, a new stack frame is typically allocated for saving the address, but since we are performing generic call mediation, a new stack frame cannot be allocated due to the fact that the generic mechanism must remain invisible to the caller and callee. The mediator also may not adjust the stack pointer because the callee may refer to arguments provided by the caller relative to the stack pointer set by the caller. The common mediation routine also needs to save the pointer to the mediator rep across the forwarded call invocation without using the stack. In our implementation, a non-volatile register is used to store the pointer to the mediator rep. Its original value of this register, along with the return address of the caller, and the thread identifier of the caller, are saved in a hash table by the mediation prolog, before the call is forwarded. The mediation prolog checks the current switch phase, and decides whether it should *i*) track the call in the hash table, *ii*) block the call, or *iii*) forward it to the rep corresponding to the new object. The first code path requires epilog processing for accounting purposes, while the remaining paths can skip the epilog.

When the object reaches the point where there are no in-flight calls, data transfer is initiated to provide the new object with the state required for continued consistent operation. Once the transfer completes, the new implementation is ready to accept requests. The mediator swings the pointer at the root table to point to the new root, and flushes the affected object translation entries. The pending requests are then unblocked and the mediation routine forwards the calls to the reps of the new implementation.

Walk-through of a mediated clustered object invocation

To demonstrate more concretely the algorithms described above, we present the pseudo-code walk-through for a typical call mediation with the mediator representative already installed in the OTT entry.

The client invokes the method `foo` of a clustered object referenced by the identifier `i` (i.e., `DREF(i)->foo()`, where `foo` is method number k in the clustered object's vtable). During the switching period, the mediator representative handles this call. This call made to the mediator representative ends up calling `mediatorMethod`k, a trampoline method that records the vtable offset and jumps to the common mediation routine, `mediatorMethodCommon`, which:

- first saves the argument registers[3] and the return address on stack;
- prepares the arguments for calling the mediation prolog. The arguments include the `this` pointer that refers to the actual representative to service the request (output parameter), the method number (input parameter), the return address `ra` and the non-volatile register `nvreg` (input parameters — these registers are used by the mediation code);
- saves the mediator pointer in `nvreg`;
- invokes the mediation prolog (see next paragraph for details); which
  - performs the phase-dependent prolog;
  - writes the pointer to the actual rep object (the one that will be used to service the request) to the `this` pointer;
  - sets a flag indicating whether epilog is necessary;
  - returns the real virtual function address of the representative object that is used to service the call;
- based on the flag returned by the prolog, decides whether the forwarded call should return to this assembly routine (which is the case only if we are forwarding to the

[3]This, combined with maintaining the state of the stack at the point of forwarding the call to the real method, allow the common mediation routine to work generically over all method signatures.

original object);

- if so, it
  - resets the stack and registers (to the values before invoking the prolog) and then *jump-and-link* to the real virtual function (the return address is then set to get back here);
- if not, it
  - resets the stack and registers, restores the original return address, and *jump* to the real virtual function (call is forwarded transparently and will return directly to caller);
- prepares for the epilog (for the calls that do return) by saving the result registers on the stack;
- call the mediation epilog (see next paragraph for details), which:
  - obtains the pointer to the mediator from the `nvreg` and executes the phase-dependent epilog code;
- restores the registers (including the `nvreg`) and the return address and return to caller (jump to the `ra`).

Mediator prolog/epilog pseudo-code

Here we provide the pseudo-code for the mediation prolog and epilog routines. They are called by the common mediation assembly routine, `mediatorMethodCommon`.

Mediation prolog

- PhaseLock.acquire();

```
• if (in Forward phase)
    – /* increment before releasing lock to ensure we don't reach Completed phase prematurely */
    – increment call counter;
    – PhaseLock.release();
    – therep is assigned to point to the representative from the original clustered object;
• else if (in Block phase)
    – /* insert into list before releasing lock to ensure that: 1) we don't forget to unblock it due to race, 2) we can execute hash table lookup outside the lock */
    – insert TID (Thread ID) in blocked_list;
    – PhaseLock.release();
    – look up TID from hash table;
    – if (TID in hash table)
        ∗ /* re-entrant call; do not block it */
        ∗ remove TID from blocked_list;
        ∗ increment call counter;
        ∗ therep is assigned to point to the representative from the original clustered object;
        ∗ /* note that the above count will be positive before the increment, due to re-entrancy — do not need to worry about races that will cause a phase change */
    – else
        ∗ block thread;
```

    * */* this thread will be unblocked upon exiting Block phase */*
    * indicate to `mediatorMethodCommon` that we do not want to execute epilog;
    * `therep` is assigned to point to the representative from the new clustered object;

- else if (in *Completed* Phase)
  - PhaseLock.release()
  - */* this call came in after the last mediated in-flight call exited and all blocked calls were unblocked already — just need to get the new rep pointer */*
  - indicate to `mediatorMethodCommon` that we do not want to execute epilog;
  - `therep` stores the representative from the new clustered object;
- modify the output parameter for the `this` pointer from `therep`;
- compute and return the function pointer of the real method for call forwarding.

Mediation epilog

- decrement call counter;
- if (in *Block* phase)
  - if (call counter is 0) then execute switch completion code;
- pop off the thread data from the hash table: `ra`, `nvreg`.

4.4.2 Deadlock Avoidance

One of the main challenges of implementing a generic mechanism for switching clustered object on the fly is deadlock avoidance. Specifically, we need to ensure that the threads blocked by the switching layer will eventually be unblocked when there are no more in-flight calls within the object that could potentially affect the state of the object. In particular, re-entrant calls into the object must not be blocked. As stated earlier, if these calls are blocked, the in-flight call count will stay positive and the unblocking will never happen, hence causing deadlock. Most of the re-entrant calls can be detected by the switching layer's thread identifier hash table, which keeps track of the in-flight calling thread identifiers. Calls with the thread identifier found in the table are not blocked and are forwarded to the original object so that the threads may eventually finish executing the methods within the object and terminate.

Using the hash table to keep track of in-flight calls is not sufficient alone, however, since the re-entrancy may be indirect. A thread executing a method of a clustered object may create another thread that invokes methods of the clustered object again. If the original thread waits for the created thread to terminate before continuing, and the created thread is blocked by the switching layer, deadlock will result. Since the newly created thread has a new thread identifier, it will not be detected by the lookup table. Since this situation does arise in programming distributed implementations of a clustered object[4], the switching layer should not block threads of such nature.

We address this case in the following way: in K42, every thread can be checked whether the thread was created explicitly (using the multiprocessor-messaging library or the `Scheduler::ScheduleFunction()` method) or created implicitly by the protected procedure call (PPC) facility in response to a cross-address-space external object method invocation. Indirect re-entrancy only occurs when threads are created explicitly. Hence, to protect from the aforementioned deadlock situation caused by indirect re-entrancy, during the *Block* phase, if the calling thread is determined to be an explicitly created thread, then it is not blocked, but instead forwarded as if it is a re-entrant thread. If the thread is created implicitly to handle an external PPC request, it will be blocked in

[4]While this is not the most common case, it is conceivable that a multi-rep clustered object may administer the reps by means of creating one worker thread for each representative and determine the threads' completion by using a barrier. If those threads invoke the clustered object recursively, and if the switching layer blocks these threads, deadlock will occur.

the way we described above. This scheme can successfully prevent deadlock caused by a chain of program-spawned threads that indirectly creates re-entrancy, since those threads are never blocked by the switching layer. In other words, it avoids the deadlock problem by detecting that a calling thread *could* cause indirect re-entrancy and not blocking it, just to be safe.

A consequence of letting explicitly created threads be forwarded instead of blocked is that it may take longer for the in-flight call count to reach zero. Also, it is possible to construct an infinite sequence of thread creations and clustered object invocations in such a way that it will keep the in-flight call count from ever reaching zero, hence disabling the switching layer from switching the clustered object. However, this should not pose a problem in practical clustered object implementations.

To minimize the possibility of "live-locking" the switching progress, implicitly created threads (external PPC threads that are not re-entrant) are blocked during the *Block* phase. One might question the appropriateness of blocking these calls in face of the indirect recursion problem: it is possible also to have a chain of external PPC method invocations that leads to a call cycle, and blocking such call after a cycle will then cause deadlock. While it may still be possible to encounter deadlock if the indirect recursion happens through an external PPC chain, we decided to leave it as the responsibility of the system object programmer to prevent cyclic external PPC call chains since such programming practice is not recommended in any case, as it may lead to other more fundamental problems such as re-acquiring of per-object shared locks. In general, the system object programmer should bear in mind that when an external PPC is made to another server object, that call may be blocked, or even fail.

4.4.3 Multiprocessor implementation issues

We now describe some of the multi-rep mediator implementation issues that arise in more detail. Since the phase variable associated with the switch operation is checked by all mediated calls, and a lock is needed to coordinate the accesses, we implemented the mediator as a fully distributed clustered object with one rep per processor to achieve good scalability. Each rep maintains its own local data, such as the pointer to the original rep that it should forward the calls to, the mediated call count, the hash table, a local switch phase variable, and the phase variable access lock. The root maintains a phase variable also, but it is accessed less frequently and used only to perform proper phase-dependent miss-handling. This way, the common path of call mediation will not need to acquire and release a global lock. Only the per-representative lock is held and only the local switch phase is examined when a call enters the mediator to be forwarded or blocked in the common case.

Switch phase variable The local phase variable determines the kind of mediation necessary to provide forward progress. The *Forward* phase implies that there may still be in-flight calls on that processor that are not yet recorded by the mediator. The worker thread that performs the generation check changes the local phase to *Block* when it determines that all the threads that could potentially be making in-flight calls to the rep running on the one processor have completed. For the most part, the phase variable can be maintained locally on a per-representative basis. However, in order to determine when to transfer state from the original to the new object, we must ensure that there are no in-flight calls for the object across all the processors of the system. That is, all the local phases are in *Block* state and that the in-flight call counters are globally zero. To determine if all the mediator reps are in the *Block* phase, we maintain a counter at the mediator root to keep track of the number of mediator reps are still in the *Forward* phase. Once that count reaches zero, we can change the global switch phase to *Block*. This global switch phase, along with the in-flight call counter (described next), are used to determine if we can carry out data transfer and change the phase to *Completed*.

In-flight call counters Each mediator representative has a local counter value that is updated whenever a mediated call is being forwarded to the original object (as a tracked in-flight call). It thus maintains the number of in-flight calls made via the local representative. The counter is only checked for zero across all processors after all the mediator reps have entered the *Block* phase. So, prior to the block phase, the local counters are updated completely independently. A shared value is maintained by the mediator root to count the number of mediator representatives that have in-flight calls. The value is lazily updated; the value is only maintained by a rep when its local phase changes to *Block*. This value does not need to be accurate until all the mediators have entered the *Block* phase, at which point the value is zero implies that there are no more in-flight calls.

Hash tables The per-mediator-rep hash table stores call forwarding data (return address, etc.) using the caller's thread identifier as the lookup key. The purpose of the hash table is to provide a location for storing forwarding information (that cannot be stored on the stack) and to provide a means to detect calls that are directly re-entrant. To detect direct re-entrancy, a local, per-representative hash table is sufficient, since K42 threads are not migratable.[5] Data with the same thread identifier key may be inserted multiple times into the hash table (in the case where the calling thread is re-entrant), so the item insertion and removal has to be in the last-in-first-out (stack) order. Also, per-hash-bucket list locks are used to further reduce potential shared-lock contention.

Lazy mediator-rep creation The installation of a mediator follows the clustered object rep creation paradigm, where the mediator root is created but its reps are created on first use. While this avoids unnecessary rep creation, it also has complications. The main issue is that even though no new calls are made on the processor, the mediator

[5]When threads become migratable in future versions of K42, some work will be needed to move the associated items in the hash table to the remote location.

worker thread will still need to run to perform generation checks and phase changes if the processor has had a rep installed prior to switch. As a result, the phase variable and the mediator call counter may exist for the processor while a mediator rep is not yet created on that processor. Therefore, these data items are maintained directly by the mediator root, in a cache-friendly manner (i.e., data for different processors are located on different cache lines).

4.4.4 Data Transfer

The data transfer method must be provided by the object designer as a callback method for the mediator. It is called by the mediator when the switching layer determines that there are no more in-flight calls in the original object (and hence the objects can be safely switched). The transfer is handled individually by the object involved in the switch. Currently, to obtain maximum throughput, the data transfer method has full access to both objects' internal data structures.[6]

In the current implementation, data transfer is performed in a somewhat ad hoc manner. As a result, each of n building blocks may have to be able to transfer data to any one of the $n-1$ other building blocks, resulting in n^2-n distinct instances of data transfer methods. While it is acceptable when the number of implementations of a building block is small, it is desirable to have a more structured way of handling the data transfer problem for the general case. Also, it may not always be possible for the data transfer method to obtain full access to both objects' internal implementations. More generic and automated mechanisms for performing data transfer will therefore be necessary.

The primary goal of the data transfer method should be to eliminate the requirement of each building block implementation having to understand the internal data structure

[6]An example of such implementation is a transfer method that is declared to be a `friend` by both objects.

of the other building block implementation for data transfer. This can be achieved by negotiating data transfers through a well-defined set of transfer interfaces. Minimally, a standard class serialization interface can be provided. For each building-block interface I, there is a canonical serialization interface T_I^0 that is common to all the building blocks implementing I. Often this transfer function is based on the abstract data type that is being exported. This approach avoids the need for one data transfer method per each pair of building-block instance. However, this lowest-common-denominator canonical serialization interface has the potential disadvantage of inducing unnecessary data transformations that can be avoided given a more optimized form of data transfer interface for the particular pair of building block implementations.

An additional set of more optimized data transfer interfaces $\{T_I^1, T_I^2, T_I^3, \ldots\}$ can be developed. An interface-specific transfer method would then carry out negotiations for the building blocks to be switched and determines the common transfer interface supported by both building blocks to perform data transfer in the most efficient way possible. For instance, suppose that building block B_1 is being switched to B_2. B_1 provides data transfer interfaces $\{T_{I_B}^0, T_{I_B}^1, T_{I_B}^2\}$ while B_2 can use $\{T_{I_B}^0, T_{I_B}^1\}$. Then the data transfer method for interface I_B should determine that $T_{I_B}^1$ is the best available data transfer interface to be used for this switch. Note that the requirements for the old and new building blocks are different; B_1 needs to *export* the interface while B_2 needs to *apply* the interface to perform the transfer. Since a canonical serialization interface $T_{I_B}^0$ is available, the two sets of supported interfaces is guaranteed to be non-disjoint.

For abstract data types that behave as a look-up table, the data transfer function can be implemented as a simple chain-up of the old object to the new object. This way, a more lazy transfer approach can be applied (i.e., lookup and migrate data on demand). For requests to look up an item, the new object's data structures are navigated first. If the item is not in the new object, the old object is also queried. If it is found in the old object, the item is then migrated to the new object for future look-ups. The old object

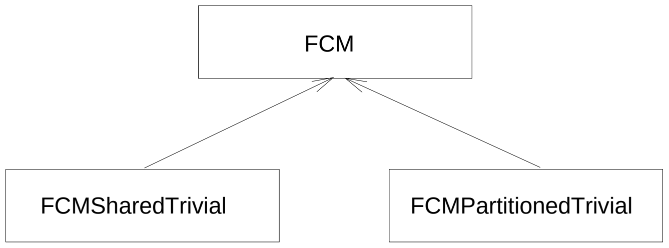

Figure 4.9: *A subset of the FCM class hierarchy.*

is deleted when all items have migrated away.[7]

Example: FCM

We use the File Cache Manager (FCM) interface to demonstrate the options available for performing data transfer between different implementations. The FCM is used to lookup page frames that are cached in memory given a file offset to identify the page frame. A variety of FCM subclasses can be implemented to optimize for different usage patterns. For example, we have a very lightweight version, `FCMSharedTrivial`, that uses a single representative with a single shared lock and a shared linked list to maintain the list of page frames. This shared implementation is best for the case where the number of pages cached by the FCM is small and when the pages are accessed by a single thread of execution. However, when there are concurrent accesses to the object, the shared lock gets contended very quickly. Therefore, to optimize for concurrent accesses, implementations with multiple representatives and localized locks are available. One of them, `FCMPartitionedTrivial`, partitions the range of page offsets among the representatives. The simplified class hierarchy is illustrated in Figure 4.9. We will use `FCMSharedTrivial` and `FCMPartitionedTrivial` to illustrate the data transfer mechanisms described above.

Both FCM implementations support the FCM building-block interface, so it is possible to apply the switching mechanism to change a running instance to the other on the fly.

[7]This is not guaranteed to happen eventually, however. A data-migration thread can be implemented to either 1) discard these items if permitted by the semantics of the object, or 2) perform the migration asynchronously when the system load is low.

```
class FCMDataTransferCanonical {

public:

    PageDesc *getFirstPageDesc() = 0;

    PageDesc *getNextPageDesc(PageDesc *curr) = 0;

};
```

Figure 4.10: *Canonical FCM data transfer interface.*

Assuming that the application program opened a small file and did not specify its access pattern, then the operating system would first create an instance of `FCMSharedTrivial` to maintain the cached pages of the file. When the access pattern indicates that there are concurrent accesses to partitioned regions of the file, the `FCMPartitionedTrivial` implementation would then be instantiated to replace the original FCM instance, and a switch would be initiated.

Let us consider the different approaches we can take to perform the data transfer from `FCMSharedTrivial` to `FCMPartitionedTrivial`. Minimally, each FCM building block supports the FCM canonical data transfer interface, `FCMDataTransferCanonical` (Figure 4.10), which relies on the fact that abstractly, a page cache contains a set of <file offset, physical page descriptor> pairs (`PageDesc`'s). `FCMDataTransferCanonical` supports getting the first `PageDesc` out of the set using `getFirstPageDesc()`, and iterating through the rest using `getNextPageDesc()`. It is a natural lowest common denominator for transferring data between FCM instances since each FCM conceptually contains a list of page descriptors which can be walked through using a simple iterator.

In addition to `FCMDataTransferCanonical`, `FCMPartitionedTrivial` supports another data transfer interface, `FCMDataTransferPartitioned` (Figure 4.11), which performs the transfer of page descriptors in a pre-partitioned manner, hence eliminating partitioning re-calculation. This can be useful if we are transferring states between two implementations, both of which are using the same data partitioning scheme.

However, since `FCMSharedTrivial` does not support `FCMDataTransferPartitioned`,

```
class FCMDataTransferPartitioned {

public:

    SysStatus getPartitionInfo(PartitionInfo &info) = 0;

    Partition *getFirstPartition() = 0;

    Partition *getNextPartition(Partition *curr) = 0;

    PageDesc *getFirstPageDesc(Partition *part) = 0;

    PageDesc *getNextPageDesc(PageDesc *curr) = 0;

};
```

Figure 4.11: *Data transfer interface for Partitioned FCM implementations.*

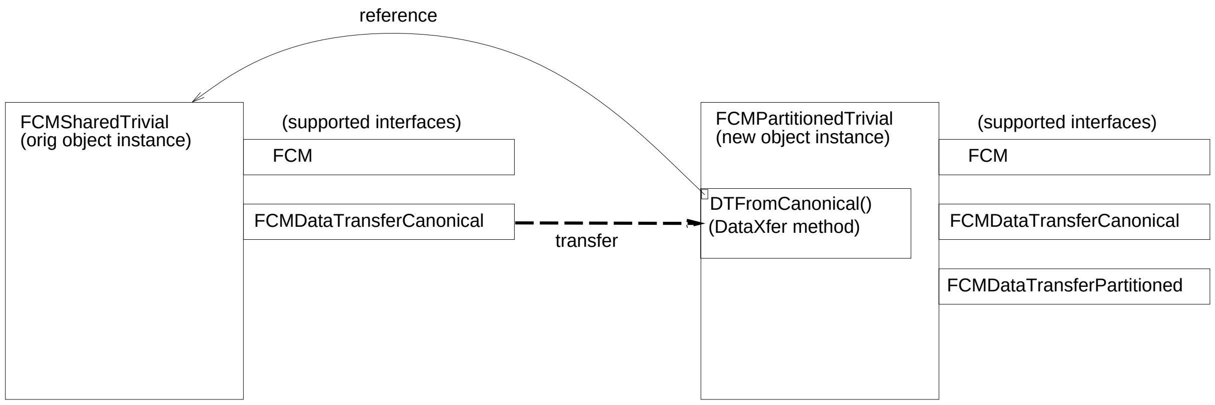

Figure 4.12: *Data transfer from* `FCMSharedTrivial` *to* `FCMPartitionedTrivial`*. After negotiation the data transfer method* `DTFromCanonical()` *is used.*

the data transfer interface used would be `FCMDataTransferCanonical`. Figure 4.12 illustrates the data transfer between `FCMSharedTrivial` and `FCMPartitionedTrivial`.

As an optimization, in the particular case of FCM building blocks, a lazy transfer protocol can be applied for the FCM building blocks. The transfer method would simply cache the reference to the original `FCMSharedTrivial` instance when switching over to the `FCMPartitionedTrivial` instance. Future lookups made via the FCM interface will be sent to the `FCMPartitionedTrivial` object, which will query the original `FCMSharedTrivial` object if the page is not found. If the page is found in the original object, then the page descriptor will then be migrated over to the new object. Once the original object no longer caches any page frames, the linkage is then removed and the original object freed.

4.4.5 Implementation Status

The mediator clustered object, including the transparent call forwarding mechanism, is implemented and running on both the MIPS and PowerPC platforms. The dynamic switching infrastructure is built on top of the existing K42 clustered object facility. It is implemented using about 2200 lines of heavily commented C++ source code and about 400 lines of assembly. The current implementation for data transfer is crude; the more generalized form using canonical transfer interfaces has not yet been implemented.

4.5 Summary

We described the design and implementation of the dynamic clustered object switching infrastructure. This facility aims to provide dynamic customizability to K42's building-block composition. It leverages K42's support for clustered objects and thread generation counting mechanism. The clustered object system provides true separation of interface and implementation (including internal data distribution), thus providing a flexible framework for interposing and switching object implementation on the fly without affecting the clients. The generation counting mechanism provides a simple and efficient way for the system to determine if there are still threads alive within a process, and this is used by the switching layer to determine if there are still in-flight requests executing within an object. This knowledge allows the facility to switch object implementation even when the object is alive and busy servicing requests via the clustered object interface.

The switch is accomplished by introducing a generic mediator clustered object which accepts new requests in place of the original clustered object, regardless of the interface exported. The take-over can be performed easily with the indirection available from the clustered object system. The mediator forward calls back to the original object until the original object's non-mediated in-flight requests are all finished, at which point new calls are blocked and the mediator waits for the mediated forwarded calls to complete. Subsequent to the completion of the mediated forwarded calls, relevant object state is transferred to the new implementation and the new object then takes over, handling the blocked and new incoming calls. The mediator clustered object is implemented as a multi-rep clustered object to provide good performance and scalability in a multiprocessor environment.

Chapter 5

Performance

In this chapter we examine the performance of dynamic switching in K42. First we give a brief discussion of the experimental framework we used. Then, we explicitly identify the costs associated with the dynamic switching mechanism. We also describe the runtime costs of using this mechanism to perform on-the-fly customization. Lastly, we present a set of experimental results showcasing the performance advantage one can achieve with dynamic object switching.

5.1 Experimental Framework

K42 currently supports both IBM RS/6000 servers and the University of Toronto NUMAchine platform [31]. Our experiments were run on an IBM S70 enterprise server with 12 PowerPC RS64 processors clocked at 125.9 MHz and a 4MB unified L2 cache. The machine's bus speed is 83 MHz.

We also ran our experiments on a NUMAchine configuration with 16 MIPS R4400 processors clocked at 150 MHz, each with 16KB direct mapped L1 data and instruction caches and a 1MB unified L2 cache. The processors are organized in stations of four processors with a memory module per-station and the stations are interconnected by a

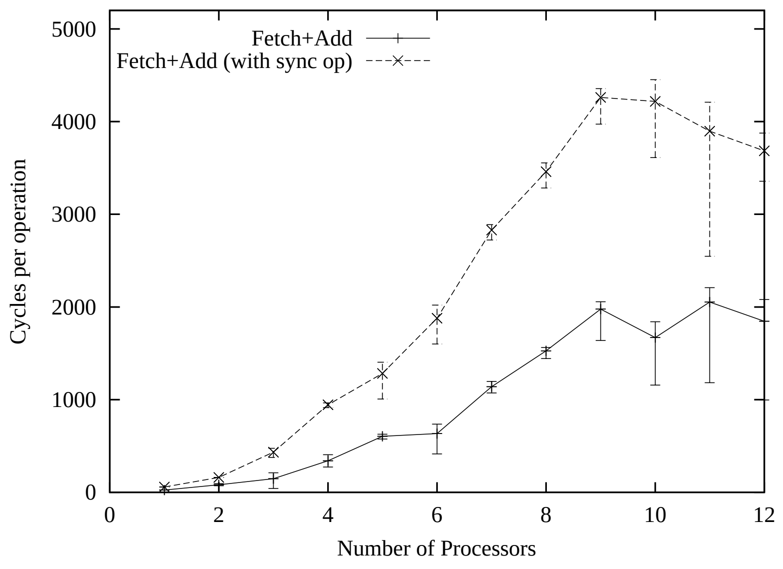

Figure 5.1: *Costs of updating a single shared variable.*

ring.[1] The results gathered on NUMAchine reflect the same trends as those of the S70, however the larger ratio between processor to bus speed and NUMA effects result in greater sensitivity to locality and better demonstrate K42's scalability. But since the results are similar, for simplicity we present the results from the IBM S70 platform.

To motivate and provide insight into the impact of sharing on the S70 hardware platform, we present, in Figure 5.1, the results of two simple tests which update a single integral variable. We ran the program multiple times varying the number of processors on which it spawned threads. The interval bars on each point indicate the range of results from all the test threads, and the main point is the average.[2] Each thread performs 100000 successive updates of the shared variable. The variable was updated with an atomic load-linked store-conditional (LLSC) instruction. In one experiment, we followed this with a memory synchronizing instruction (`sync`), and, in the other, we used just the LLSC. On processors such as the PowerPC with a weakly consistent memory model, a memory

[1]There are actually two levels of rings, local and global. However, since the experiments are restricted to 12 processors, the global ring is not used.

[2]All the graphs in this chapter have these vertical interval bars; some bars are so close to the main points they are unnoticeable.

Fixed Per-switch Costs	
Item	Instructions
Miss-handling cost	344
Mediator rep instantiation	838

Fixed Mediation Costs	
operation	Instructions
Rep method invocation (inc)	32
Forwarding to old obj via mediator	633

Table 5.1: *Costs associated with call mediation.*

synchronizing instruction is needed to ensure that at critical points the consistency of memory is maintained. Locks intended for a multiprocessor must also use the `sync` operation. The results of Figure 5.1 demonstrate that in the worst case on 12 processors, the simple updating of a shared variable can cost two orders of magnitude[3] more than the base uniprocessor cost even without a memory synchronizing instruction.[4] Including memory synchronizing instructions doubles the costs of updating the contended variable.

5.2 Dynamic Switching Costs

The costs associated with switching can be separated into two components: a set of fixed costs inherent to the mediation process, and a workload dependent component.

5.2.1 Fixed costs

Table 5.1 reports the cost associated with call mediation during the interval between switch initiation and completion. The mediation overhead associated with forwarding an object method invocation to the original object prior to state transfer is about 633 instructions. This involves 1) the register saves and restores (once for the prolog and once for the epilog), 2) the mediator prolog (phase check and hash table insert), and 3) the mediator epilog (the hash table retrieve and delete). While this overhead is non-negligible, the cost is incurred only for those method invocations that take place during the *Forward* phase. After the *Block* phase, the invocations that are redirected to the new object do not perform hash table operations nor do they execute epilog code.

The cost of the first mediated call after a switch initiation is about 1815 instructions. This includes 344 instructions for the clustered object miss-handling invocation, 633 instructions for the call forwarding (described above), with the remainder (838) attributed to the instantiation of the mediator representative.

In our multiprocessor implementation of the mediator clustered object, a thread is spawned on each processor with a valid OTT entry for the object to be switched. This thread resets the OTT entry and maintains a local switch phase variable used by the mediator. The thread is also responsible for polling the thread generation count periodically to see if the old threads have completed. The cost of executing the worker thread is about 2786 instructions (assuming only one pass through the polling code).

5.2.2 Variable costs

In addition to the mediation costs, there is an object-dependent cost of performing the data transfer which the implementor of the object being switched is required to provide.

[3]The uniprocessor case takes 24 cycles while the 12-processor case takes an average of 1846 cycles.

[4]The dip at the high end of the graph may be explained by a convoy situation created due to the simplicity of the fetch and adds.

For the switch to complete, the switching code needs to determine when there are no more in-flight calls within the object. This is done by thread generation checking. During the *Forward* phase (i.e., before the generation in the local processor is elapsed), calls are forwarded to the old object. Once the generation is elapsed new calls are blocked until all the tracked in-flight calls are completed (the *Block* phase). Only when all the worker threads have passed the *Block* phase can data transfer occur and switching complete.

The time that it takes for the phases to change from *Forward* to *Complete* is dependent on the workload of the system. In a very active system, with large numbers of in-flight requests, the switch may take longer. Another factor that limits the switching time is how long it takes to process the requests already in flight, which is generally short in K42 kernel objects.

5.3 Experimental Results

In this section we present experimental results for an isolated counter as well as Region and FCM objects that are used in K42's memory management code.

5.3.1 Description of the Counter Objects

In order to gain insight into the performance of dynamic switching, we implemented two versions of counter object, one optimized for concurrent reads (to obtain the value of the counter), and the other optimized for concurrent updates (to increment or decrement the counter).

As we saw in Figure 5.1 a simple shared counter will not perform well when it is frequently updated on a multiprocessor. Therefore, a more scalable implementation is needed for the counter optimized for updates. To do this, we implemented a partitioned counter that ensures that only local accesses are required for updates. Each processor has its own representative with a separate counter value. Locking and updating occur

```
Enter barrier
Record start time
Loop for 100000 times
    Update counter
    Yield thread
Record the end time
```

Figure 5.2: *Pseudo-code for counter test threads.*

independently on each processor and do not require global synchronization. However, in order to ensure that we maintain the semantics implied by the simple shared counter, we need to lock all the individual counters to sum the values. For the counter optimized for reads, a shared implementation is sufficient and has the advantage that an atomic primitive can be used for updating, avoiding the need for a lock.

5.3.2 Counter Experiment

We ran three simple multi-threaded programs that used these counters. The `update` experiment tests a counter with a series of updates by spawning off between 1 and 12 threads that execute the code in Figure 5.2.[5]

The `read` experiment is identical except that the update is replaced with a read of the counter's value in the loop. The `two-phase` experiment essentially combines the `update` and `read` experiments; it goes through an update and then a read phase with a barrier between them. For this last experiment, we start the test counter optimized for the update access pattern and then switch it to the implementation optimized for the read access pattern. The switch was invoked after the threads start the second phase (the read phase). The switch is initiated by calling a switch-initiation method of the original object. The method instantiates the new implementation and uses the dynamic switching

[5]When we ran these experiments, K42's preemption support was not complete, requiring us to explicitly yield the processor to other threads (in our case to mediator worker threads).

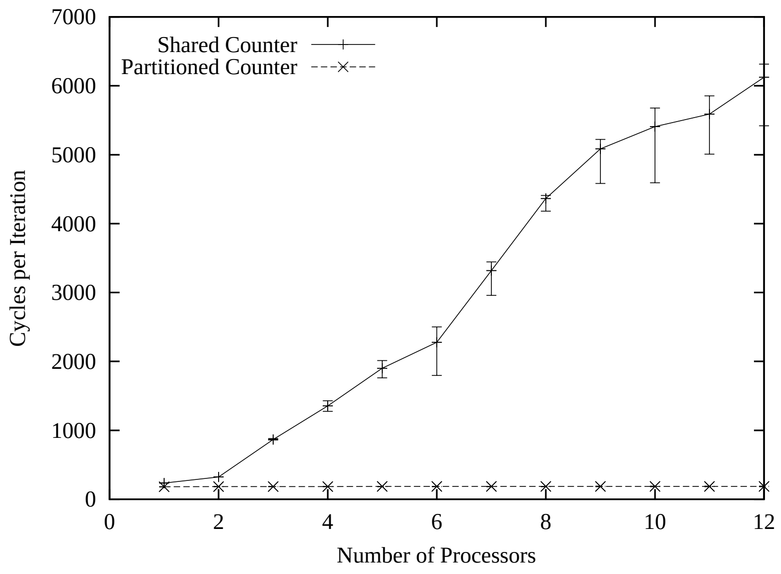

Figure 5.3: *Counter update performance comparison*

layer to switch to this new implementation, backing the counter clustered object reference being tested.

5.3.3 Results of Counter Experiments

Figure 5.3 illustrates the results of testing both versions of the counter with the update access pattern. As expected, we see that the counter implemented with a shared variable has similar performance as the experiment of Figure 5.1 and has poor scalability. It is interesting to note that the slight delay introduced by the thread yield in the counter test program is enough to break the convoy effects see in Figure 5.1 resulting in a smoother curve with consistent worsening performance up to 12 processors. On the other hand, the partitioned counter is able to achieve the same performance from 1 to 12 processors as the work done per-update is independent and constant. The partitioned implementation is the better choice for the update access pattern.

Conversely, in Figure 5.4, we see that the shared counter performs better. It is easy to understand that in the case of reads the shared counter's value will remain in the cache

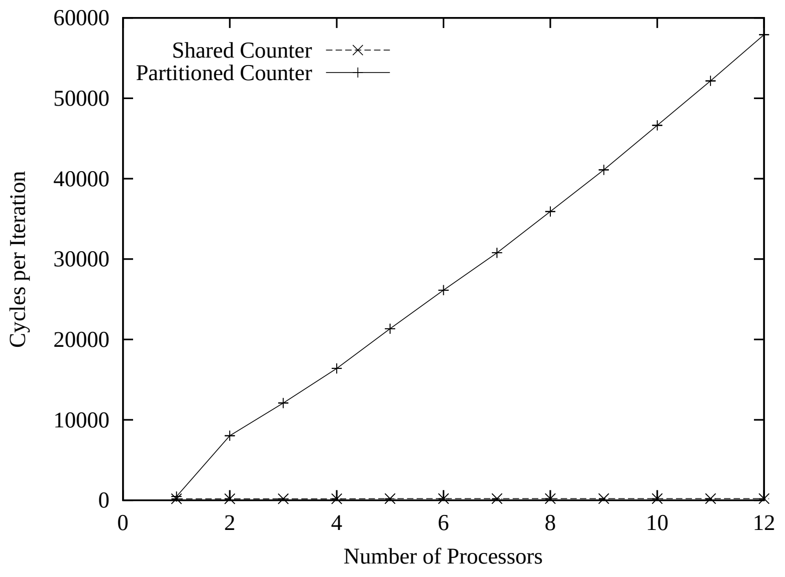

Figure 5.4: *Counter read performance comparison*

of all the processors which access it and hence will require a constant and independent cost for each access. On the other hand, the partitioned counter requires considerably more work. In order to obtain the global value each individual lock must be locked and then all values summed.

To understand why the values are worse than experienced with the shared counter under the update access pattern, we must probe deeper into the implementation. Unlike the shared counter, each local counter value of the partitioned counter must use an explicit lock and cannot rely on the use of an atomic primitive due to the need to synchronize across all the local counter values. We used blocking locks which, when uncontended, do not exhibit increased costs. However, when contended, blocking locks exhibit considerably more overhead than simple atomic primitives (which essentially implement a spin). At 12 processors, accessing data in a shared location via a contended lock is more expensive than using atomic primitives to update that data. Further, the extended period of time the locks are held in the partitioned counter further exacerbate the contention.

As expected and shown in Figure 5.5, in the `two-phase` test we see that neither the

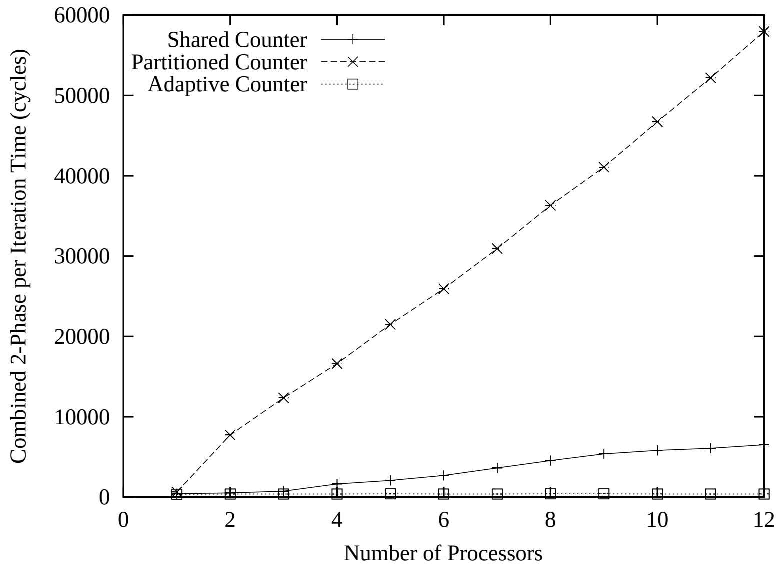

Figure 5.5: *Performance of different counter object implementations*

shared nor the distributed implementations are able to meet the requirements of both access patterns. However, if we dynamically switch counters between the two phases, the individual advantages of the different counters are exploited for the appropriate phase, achieving better overall performance.

5.3.4 Description of the K42 Objects

As described in Section 3.2, a number of building blocks are used to provide memory management services in K42. We focus on two objects: the Region object and the FCM object. Each building block has a simple shared implementation and more complex distributed implementations intended for use on large files accessed across many processors.

The attribute of the Region objects that is relevant to this experiment is a counter that tracks requests to the Region. The shared and distributed implementations of the counter maintained by the Region objects are identical to the shared and distributed counter describe the the prior experiments. The region updates the counter frequently but only needs coordinated on its value at destruction time. The shared FCM implementation

```
Loop for 10000 times
    Enter barrier
    Record start page fault time
    Touch pages within the file but local to the thread
    Record end page fault time
    Yield processor
    Unmap pages
```

Figure 5.6: *Pseudo-code for page fault test threads.*

uses a single locked list to maintain the cached page descriptors. While not scalable, it works well under uncontended, uniprocessor access and has a small memory footprint. The distributed implementation uses a hash table per-rep with fixed sized buckets of page descriptors, a lock per bucket, and a busy bit for each page descriptor within a bucket. The distributed FCM ensures that separate portions of the file are cached by separate reps thus partitioning the page across the reps. The distributed version provides scalability but at increased uniprocessor costs compared to the shared version for small files.

5.3.5 Page Fault Experiment

The `page fault` experiment is a simple user-level program that maps a file either with the shared FCM and Region or with the distributed FCM and Region[6] implementation. The program then spawns threads that execute the code in Figure 5.6. The program induces a partitioned access of the file with each processor touching it own pages. A second version of the program which allows a switch between the shared and distributed implementations was used to explore the performance of dynamic switching. Since the uniprocessor implementation starts performing poorly whenever there is any multipro-

[6]To eliminate I/O costs all the pages were initially faulted on to ensure that they were in-core.

cessor interaction, we explicitly initiate the switch when at the end of the first iteration if there are multiple number of test threads.[7]

5.3.6 Results of Page Fault Experiment

In Figure 5.7, the distributed objects are better when the accesses occur on more than one processor since they avoid the contention that exists with the shared implementation. However, Figure 5.8, which zooms in on Figure 5.7, shows the higher cost of using the distributed implementation on a uniprocessor. Examining both figures we see that dynamic switching allows us to adapt on the fly between the uniprocessor and multiprocessor workloads achieving better overall performance.

5.4 Cost-Benefit Tradeoffs in Dynamic Switching

The switching infrastructure is intended to support switching of objects that are expected to change between long phases of access patterns that can benefit from different policies/implementations, and to facilitate live-swapping of objects for on-the-fly version upgrades. However, it is interesting to discuss the limits on when the dynamic switching can be worthwhile.

If the objective is to carry out live update of a long-running object (for bug-fix or upgrade), then the switch is worthwhile regardless of the time and cost it takes to complete the switch. On the other hand, if the intent is to adapt to changes in user behavior patterns, the dynamic switch may not be worthwhile if the pattern changes are too quick — at some point the overhead for dynamically switching among strategies will overwhelm the benefits of doing so.

To make a switch worthwhile, the cycles used in performing the switch must be

[7]For the purpose of demonstration this is sufficient. However, in practice, a contention-sensitive lock can be used to detect contention and initiate a switch.

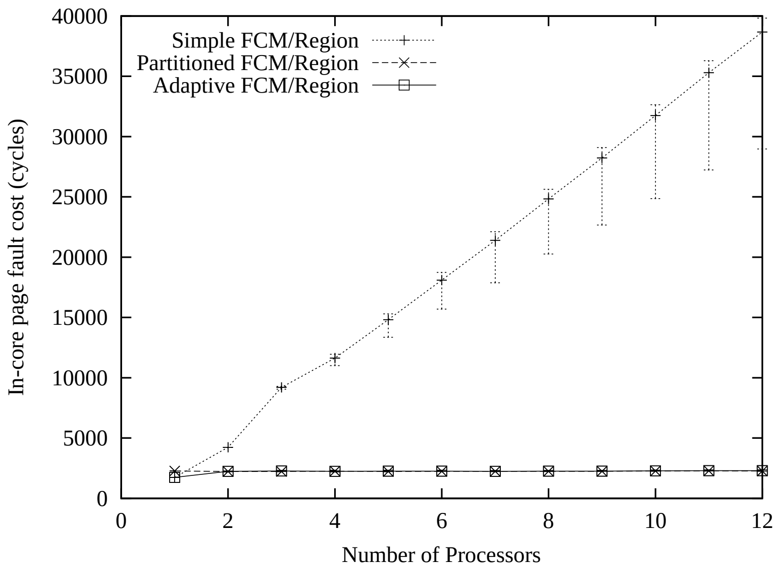

Figure 5.7: *FCM/Region performance comparisons*

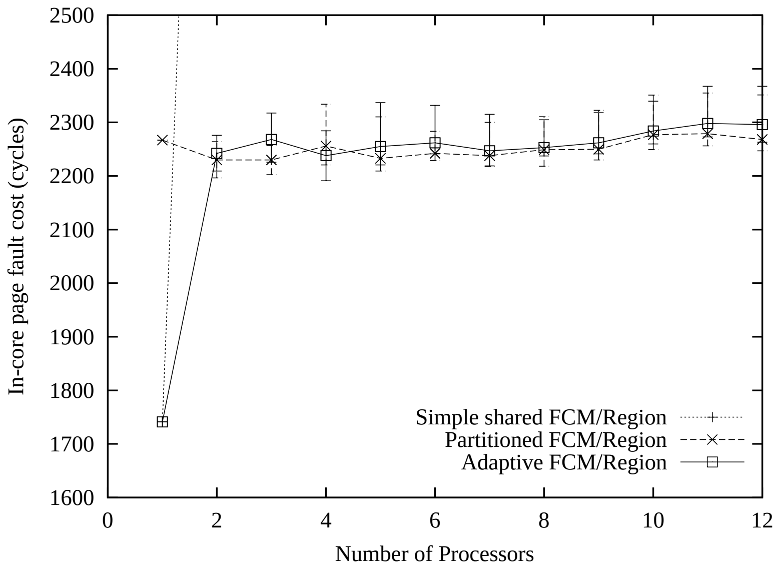

Figure 5.8: *FCM/Region performance comparisons (zoomed in)*

less than the cycles saved by using the new implementation for the duration of the new program phase that benefits from the new implementation. From the earlier cost analyses, the mediation incurs approximately $1200 + 633F$ cycles, where F is the number of mediated calls to the original object. The value of F is dependent on the workload of the system, but is generally small. Early experiments indicate that F is generally less than 5. Additionally, there are overheads from the creation and execution of the mediator worker threads, and overhead due to cache disturbances from running the switching code. It is estimated that, in general, the combined cost would be within the order of 10000 cycles. It can be as little as 4000 cycles if the switch is not complicated by in-flight calls and system load. While the cost is non-negligible, generally the benefit obtained from switching to the new implementation far outweighs the cost.

5.5 Summary

We have presented the experimental setup, explained the costs associated with the switching mechanism, and demonstrated the performance potential of applying dynamic customization to adapt to changing access patterns and resource requirements.

The costs associated with dynamic switching include mediator instantiation, miss-handling, rep creation, per-call redirection overhead, and the execution of mediator worker threads. The time it takes to complete a switch depends on the workload of the system, as well as the number of in-flight requests in the original object. Since it uses the generation counting mechanism to advance the switch phase, the more active threads the system has at one time, the longer the switch may take.

The counter experiments demonstrate that programs with different usage phases can benefit from the ability to switch implementation during execution to obtain improved performance. The page fault experiment shows how dynamic switching can improve the performance of different classes of programs, using memory-mapped files under different resource usage requirements. While simple light-weight implementations of memory management building blocks may perform better for small files in a uniprocessor program, the performance quickly deteriorates when used in a multiprocessor program. If the system does not have prior knowledge of whether the program uses the file concurrently, by using dynamic customization, it can use the simple implementations in the beginning, and switch to using more complex building block implementations when the file accesses become concurrent.

Chapter 6

Concluding Remarks

K42's achieves customizability through its building-block design. We have developed a generic mechanism that allows on-the-fly customization by dynamically switching objects. The mechanism works even while there are in-flight requests to the object being switched.

The primary goal of dynamic customization is to complement the static, creation-time customization provided by building blocks. Dynamic clustered object switching allows long-running system servers to change object implementations on the fly without rejecting incoming requests and without needing to restart the servers. This facility can be used to customize long-running services with phases of execution that have different object access properties. It can also be used to upgrade and replace older versions of object implementations in systems where maintaining an operational status is a requirement. When the original (legacy) object implementor provided the canonical data transfer method, it can in theory be switched to any future object (with the same interface) that implements any arbitrary policy.[1]

The dynamic switching facility is built as a generic software layer that can be applied by all K42 building blocks. The layer is built on top of K42's object infrastructure, lever-

[1]The cost of the data transfer may be a concern. However, if the objects are relatively fine-grained, the benefit should easily overcome the cost. This is the case especially when the object is long-lived, or when the object is required to stay online.

aging the technologies of clustered objects, protected procedure call (PPC), and thread generation model. Clustered objects further extend the object-oriented paradigm to provide the abstraction of internal data distribution in a shared memory multiprocessor for better locality and concurrency. The PPC model implies that external requests to system objects are serviced by new logical threads. The thread generation model associates a generational timestamp with each thread that is created to handle server requests. The switching layer takes advantage of the level of indirection provided by clustered objects to mediate requests to a running clustered object. It uses PPC and the thread generation model to provide an effective way to determine the existence of in-flight requests that are made to a server process, and decide when it is safe to allow the new object instance to take over.

In the results, we demonstrated that this dynamic customizability is beneficial for multiprocessor performance. In a simple counter example, where we considered the isolated effect on an individual object, we showed that mismatching the choice of data structure and object request pattern can result in significant performance degradation. On 12 processors, the shared implementation performed 300 times better for a read access pattern, while the distributed implementation performed 30 times better on a update access pattern. Dynamically customizing the counter object allowed us to achieve the optimal performance for both cases. On a combined access pattern, dynamic switching achieved an order of magnitude better performance than the shared object, and two orders of magnitude better performance than the distributed object.

We also examined how dynamic customization could impact the performance of memory management objects being used in K42. We showed that dynamically customizing the region and FCM objects allowed us to capture substantial advantages for the overall cost of a page fault. For the uniprocessor case we showed a 25 percent improvement over the object optimized for the distributed case. For the multiprocessor case, the distributed object had a large performance improvement over the centralized implementation. We were able to take advantage of both of these objects by dynamically switching as the request pattern changed.

We are at an early stage of exploring dynamic customizability in K42. We have concentrated on the multiprocessor performance advantages but expect to see advantages beyond that. Our early experience indicates there are significant programmability advantages as well as maintainability advantages.

6.1 Future Work

Now that we have a working implementation of the dynamic switching infrastructure, we can experiment more with the properties of different building block implementations under different usage patterns, and evaluate the potential benefits of performing dynamic switching between different implementations. As the operating system matures, there will be more system components available for experimentation.

We have not yet fully explored the issues involved in data transfer between the switching objects, and this may need more packaging to make it viable. In this dissertation, we have concentrated on the mechanism, not on the policies. We expect in the future to investigate the operating system automatically monitoring the system and detecting when to initiate a dynamic switch, for example, when a file grows too large, when the contention for a shared lock increases, or when its reference pattern changes. There has been other work in this area [7, 27] and we would like to be able to leverage it to help K42 self adapt. We believe that dynamic on-the-fly customization will again result in the radical performance advantages that researchers demonstrated with customizable operating systems.

Bibliography

[1] Jonathan Appavoo. Clustered Objects: Initial Design, Implementation and Evaluation. M.Sc. thesis, Dept. of Computer Science, University of Toronto, 1998.

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

[3] B. Bershad, S. Savage, P. Pardyak, E.G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th Symp. on Operating Systems Principles*, pages 267–284, 1995.

[4] R. Campbell, N. Islam, D. Raila, and P. Madany. Designing and implementing Choices: an object-oriented system in C++. *Communications of the ACM*, 36(9):117–126, 1993

[5] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. *Proc. 15th Symp. on Operating Systems Principles*, pages 12–25, 1995.

[6] K.J. Duda and D.R. Cheriton. A caching model of operating system kernel functionality. In *Proc. 1st Symp. on Operating Systems Design and Implementation*, pages 179–193, 1994.

[7] Y. Endo, Z. Wang, J. Chen, M. Seltzer. Using latency to evaluate interactive system performance. In Proc. 2nd Symp. on Operating Systems Design and Implementation, Seattle WA, October 1996.

[8] D.R. Engler, F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th Symp. on Operating Systems Principles*, pages 251–267, 1995.

[9] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, pages 137–152, 1996.

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

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

[12] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Symp. on Operating Systems Design and Implementation*, pages 87–100, 1999.

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

[14] N. Islam. *Distributed Objects: Methodologies for Customizing Systems Software.* IEEE-CS Press, 1996.

[15] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. *A fair fast scalable reader-writer lock.* In *Proc. 1993 International Conference on Parallel Processing*, 1993, pages II-201–II-204.

[16] O. Krieger and M. Stumm HFS: A performance-oriented flexible file system based on building block composition. *ACM Trans. on Computer Systems*, August 15(3), 1997, pages 286–321.

[17] I. Leslie, D. McAuley, R. Balc, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed

Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7): 1280–1297, 1996.

[18] J. Liedtke. On micro-kernel construction. In *Proc. 15th ACM Symp. on Operating System Principles*, pages 237–250, 1995.

[19] P. McKenney and J. Slingwine. Efficient kernel memory allocation on shared-memory multiprocessor. In Proc. Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA, pages 295–305, Berkeley, CA, USA, Winter 1993. USENIX.

[20] A. Montz, D. Mosberger, S. OḾalley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A communications-oriented operating system. *Technical Report 94-20*, 1994

[21] The component object model (COM): A technical overview Microsoft White Paper, Microsoft Inc., 1996

[22] P. Pardyak and B. Bershad. Dynamic binding for an extensible system. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, pages 201–212, 1996.

[23] E. Parsons, B. Gamsa, O. Krieger, and M. Stumm. (De-)clustering objects for multiprocessor system software. In *Proc. 4th Intl. Workshop on Object Orientation in Operating Systems 95 (IWOOOS'95)*, pages 72–81, 1995.

[24] C. Pu and J. Walpole. A study of dynamic optimization techniques: Lessons and directions in kernel design. *Technical Report OGI-CE-93-007*, 1993

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

[26] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, pages 213–228, 1996.

[27] M. Seltzer, Y. Endo, C. Small, and K. Smith. Self-monitoring and Self-adapting Operating Systems. In Proc. 6th Workshop on Hot Topics in Operating Systems, 1997.

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

[29] R.C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.

[30] S.R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proc. USENIX Conference*, pages 238–247, 1986.

[31] Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, and S. Srbljic. The NUMAchine multiprocessor. Technical Report 324, University of Toronto, April 1995.