INTERPOSING ON CALLS IN THE K42 OPERATING SYSTEM

by

Raymond Fingas

A thesis submitted in conformity with the requirements

for the degree of Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

Interposing On Calls In The K42 Operating System

Raymond Fingas

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2007

Code interposition allows new code to be dynamically inserted into a call path at run time. This work presents the design and implementation of a facility that allows new code to be inserted into calls to functions in clustered objects in the K42 operating system. Indirection through K42's object translation tables is used as an insertion point. Potential uses include debugging, performance monitoring, and hot swapping of clustered objects. Performance evaluation shows that the interposition facility developed has low enough overhead to be suitable for these uses.

Acknowledgements

Thanks to my fellow lab members David Tam, Jonathan Appavoo and Reza Azimi for their guidance and assistance. Also, thanks to my supervisor, Michael Stumm, for much guidance and support. Finally, thanks to the students of ece299 in 2005, for providing endless amusement and distraction.

Table of Contents

Table of Figures

Chapter 1

Introduction

Code interposition refers to dynamic alteration of program control flow at runtime so that the program executes code in a way unforeseen at the time the original code was written. Perhaps the best known application of code interposition is in the context of debuggers, when breakpoints are added to a program at runtime. When a breakpoint is reached during code execution, control is passed to code belonging to the debugger, which in turn may at some later point pass control back to the original code. Code interposition can be used for many things besides debugging, including performance monitoring, performance optimization, adaptation and integration of system updates into a running system.

Figure 1.1 shows two examples of code interposition. In the first example, interposed code is used to augment original code by adding new functionality. At a particular point in the program, control passes to interposed code, and when done, the interposed code passes control back to the place in the original code where the program's control was intercepted. In the second example, interposed code is used to replace some of the original code. When the interposed code finishes running, the original code resumes running at some point after where control was intercepted.

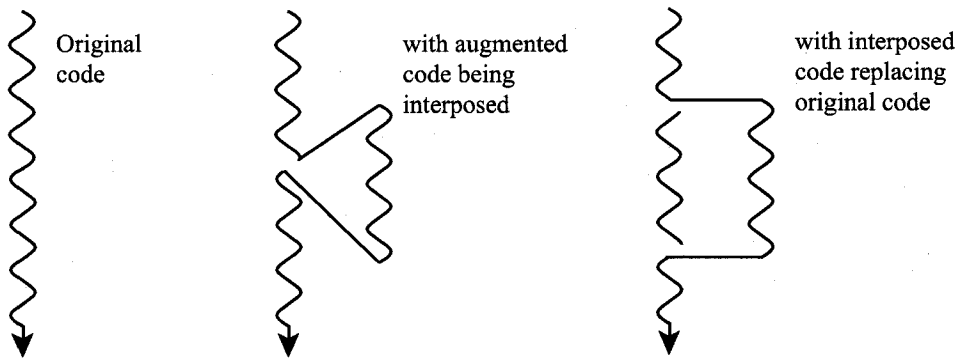


Figure 1.1: Two Examples of Interposition

Code interposition has been used in a number of systems in the past. For example, in IBM PC-DOS, new interrupt handlers were interposed by overwriting an entry in the interrupt table with a jump instruction to

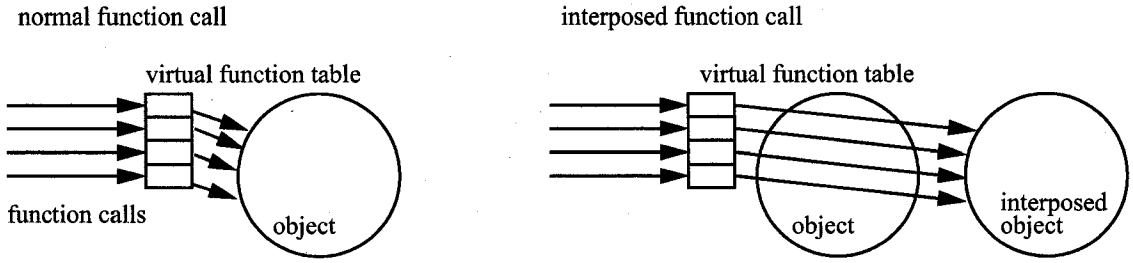


Figure 1.2: Interposing on an Object

the new handle and chaining the existing handlers onto the newly inserted handler so that the handlers would execute in sequence [12]. More recent operating systems, such as Mach, provided support for interposing on system calls by reflecting the system call back to the application for handling [1]. Some operating systems, such as SPIN, also support interposing on functions within their kernels. SPIN allows applications to download code into the kernel, and an elaborate mechanism is used to decide at runtime which one of several functions should be called during execution, depending on the currently running application [19]. Debugging and tracing systems such as KITrace, DynInst and Sun's DTrace allow the overwriting of instructions with "trampolines" that divert the flow of execution so that additional functionality can be executed [16, 8, 9].

In a system supporting code interposition, it must be possible to divert the flow of control in a way that is easy from a software engineering viewpoint, without adding much complexity or overhead. Yet, interposition can be difficult to implement in practice because it needs to be done transparently to the original code base. Often, jump instructions are placed at trampoline points, and the interposed code is required to execute or emulate the replaced instructions before jumping back to the point of interposition. However, the compiler can remove much information for optimization purposes that would otherwise make code interposition easier. It can even be difficult to determine what points in a piece of code are safe to interpose on. For many applications performance is an issue in that execution of interposed code should not introduce excessive overhead.

This dissertation describes work on the design, implementation and evaluation of a facility for interposing code at object granularity in the context of the K42 research operating system. Before outlining this work in more detail, an overview of object-oriented code interposition and K42 is presented.

1.1 Object-Oriented Code Interposition

In an object-oriented operating system, where objects have public interfaces that consist solely of virtual functions, code interposition on the virtual functions is possible by exploiting the system's virtual dispatch infrastructure. The virtual dispatch infrastructure allows functions from derived classes to replace functions with the same name in a base class. Base classes can declare virtual functions that may be superseded in a derived class or even left unimplemented until the derived class. A per-object virtual function table is often used to locate the correct version of the function specified by the most derived class. The virtual function table is an array of pointers to functions. Calls to a virtual function of an object are invoked indirectly by indexing into the virtual function table, then jumping to the function that is pointed to in the table to complete the call. This is depicted on the left hand side of figure 1.2.

Using this object oriented infrastructure, it is possible to implement code interposition at object function granularity in a straightforward way by overwriting entries in the virtual function table with pointers to alternate functions. In systems, where objects have interfaces that consist solely of virtual objects, such as K42, an entire object can be interposed on by replacing all entries in its virtual function table with a set of pointers to compatible virtual functions belonging to a different object. In that case, calls to the original object are automatically redirected to the interposed object, as shown in the right hand side of figure 1.2.

Compared to other existing interposition methods, the object oriented approach is very general in that it can be used on all objects that are part of the system without requiring specialization for each type of potential target object. It isn't limited to interposing only on system calls or only within the OS kernel. However, the object oriented approach to code interposition can only be applied in a general way if all objects use virtual functions in their interfaces.

Object-oriented interposing has an advantage over other methods of interposition. For example, interposition at the object level allows targeting of a single object instance, as opposed to all objects of a class. Moreover overhead is incurred only when a method of an interposed instance is invoked. The alternative involves inserting instructions directly into program code, which means that the interposed code is executed each time the function is invoked, regardless for which object instance. If only one object instance is of interest, then a check has to be made to determine if the code is being run on behalf of that instance or on behalf of another instance. The overhead of this check is incurred whenever the code is run, and not just when the correct object instance is being invoked. Object oriented interposing is also safer and easier to use than some other approaches where difficulties can arise if code is interposed in poorly chosen

locations.

1.2 Sample Applications of Object Oriented Code Interposition

This object-oriented approach to code interposition has a number of uses. For one, it can be used for debugging purposes. This is done by interposing a debugging object capable of performing some debugging action, such as connecting to a debugger. Code interposition can also be used to multiplex calls between multiple compatible objects to divide calls to a service between multiple objects implementing that service. Code interposition can also be used to perform temporary redirection of calls to a different object, to change the destination of a data stream, or to allow for repair of a failed software or hardware component. Two of the most exciting applications of interposing are real time performance monitoring and hot-swapping of

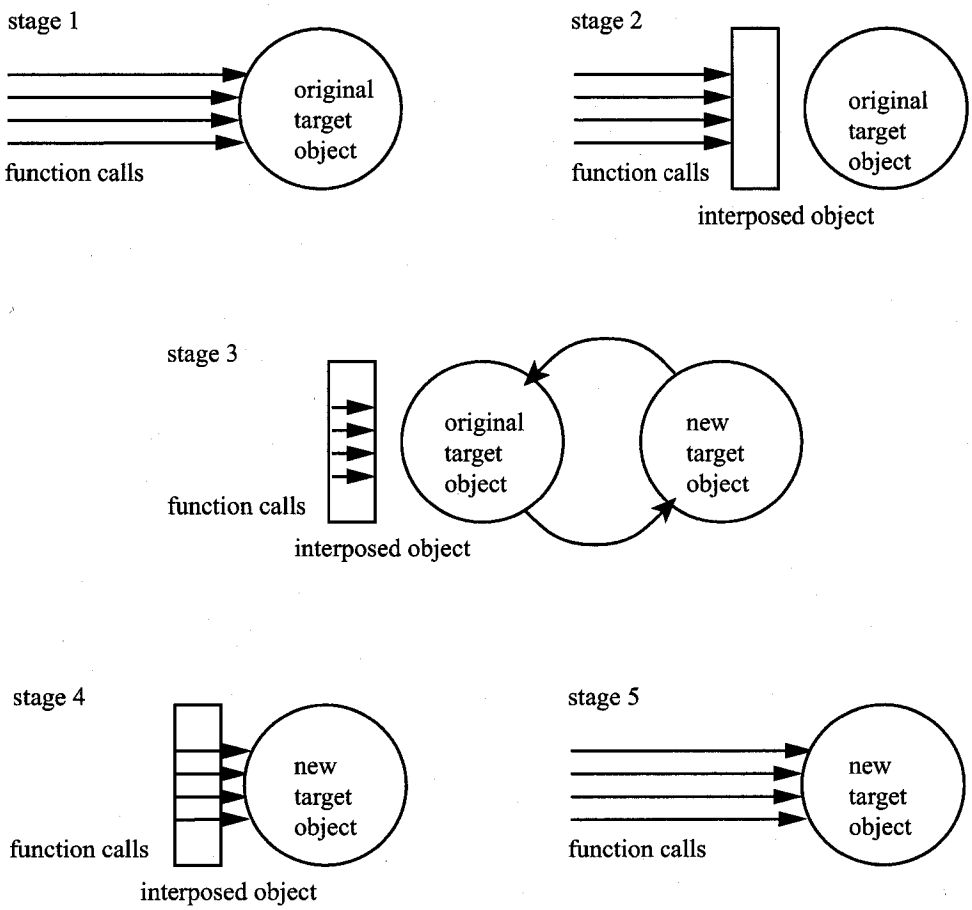


Figure 1.3: Hot-Swapping

objects within a system.

1.2.1 Performance Monitoring

Performance monitoring in real time at fine granularity is a useful application of code interposition. When it is desirable to monitor the performance of a target object, an object can be interposed on the target object that is capable of tracking performance information such as call frequency or call duration. Object interposition allows the information to be tracked with no changes to the code of the objects being monitored. That is, monitoring of the target object can be done completely transparently to the invoking code and the target object. The interposed object can track each call at a per-object instance granularity, avoiding global checks, giving it an advantage over non-object-oriented methods of interposing for performance monitoring.

Interposed objects for monitoring can potentially track many different types of performance data. Simple objects can track the number of function calls and function call timing. Other performance monitoring objects could track system statistics such as the number of page faults, system calls, or the ratio of time spent in user space to time spent executing in the kernel when a particular function was called. By using hardware counters, interposed objects could give detailed micro-architectural breakdowns of function performance, such as cache miss rates or branch predictor accuracy, as well as reasons for processor pipeline stalls [5].

1.2.2 Hot-Swapping

Another application of code interposition at the object level is hot-swapping, where an object is swapped out of a running system and replaced with a different object having the same interface [11]. Hot-swapping can be used for various purposes, including dynamic system optimization, online reconfiguration, system extensions and software updates of live systems. An interposed object be can be used to temporarily block calls so that a new object can be created, state transferred from the old object, and then the old object replaced and destroyed, before allowing the blocked calls to continue.

Figure 1.3 shows the use of an interposed object during a hot-swap operation. In stage 1, the original target object is being used normally. In stage 2, an object is interposed that blocks incoming calls to the target object. After some time, all the calls running in the original target object will have completed, and it will no longer be in use. At this point, a new target object can be created to replace the old one. The new target object must have the same interface as the old target object so that it can handle the same function calls as the old one. After being created, the new target object is initialized using the state from the old target

object. The old object is then no longer needed and can be destroyed and replaced by the new object, as seen in stage 3. After the object swapping has been completed, the interposed object allows the function calls that it had blocked to proceed. This is shown in stage 4. In stage 5, the interposed object removes itself and the system operates normally again, only with the new object instead of the old one.

There are many potential applications of hot-swapping. Some of the exciting ones include dynamic optimization, where performance monitoring is used to decide which type of object is optimal for managing a resource under the current workload, and to switch automatically to that type. As the workload changes, different objects that better handle the changed conditions can be installed. Online reconfiguration and system extensions would allow an operating system to change the tasks that it performs, or perform new tasks without shutting down the system. Software updates of live systems could allow operating system components to be updated without interrupting the system. This could enable version upgrades or patches to eliminate security bugs to be applied to a running system.

1.3 Interposition Facility and Arbiters for the K42 Operating System

This dissertation describes the design, implementation and evaluation of an object level interposing facility for the K42 operating system. K42 is an operating system that uses an object-oriented approach to achieving good scalability on large multiprocessors, and in order to benefit from the software engineering advantages that object-oriented systems have. K42 has an object system that is tailored to suit these goals. Objects that are part of the K42 object system are called clustered objects, explained in greater detail in Chapter 3. The extensive use of objects within the system and the sophisticated object infrastructure make K42 an ideal platform for implementing and evaluating interposition within an operating system and system libraries. The facility presented interposes on K42 clustered objects and inserts other objects, called Arbiters, in their place.

Arbiters are objects that can interpose on K42 clustered objects. They are called "Arbiters" because they are objects that have the ability to arbitrate function calls; that is, they can decide what actions should be taken when a function call is made, and decide if the function that was the original target of the call should be invoked or not.

The facility presented in this dissertation is capable of interposing at object instance granularity, which means that in many scenarios it incurs less overhead than other approaches that modify global code paths. This is particularly important in the context of K42 where different objects may represent different

resources, or particular instances of a service, delivered only to a specific program. Avoiding global code paths is particularly poignant on a multiprocessor, where interposing on global code paths may lead to cache lines bouncing between caches on different processors with attendant degradation of performance. The specific interposing facility described here only works with K42 clustered objects, although many of the techniques developed and observations made would apply more generally to other object-oriented interposing facilities as well.

The interposing infrastructure has been fully implemented, and several Arbiters have been written. The infrastructure supports the interposition of Arbiters around any K42 clustered object that can be paged out of memory and the removal of interposed Arbiters. Arbiters have been written that count function invocations and that time function calls. Arbiters that set a breakpoint have also been written. Micro benchmarks have been used to evaluate the overheads of interposing, and to compare interposing to a traditional debugger and to static performance instrumentation. These experiments show that the overhead of using Arbiters is acceptably low in performance sensitive situations.

1.4 Interposition Facility Requirements

In order to be useful, both in general and specifically within the context of K42, the interposing facility has a number of requirements. In addition to being able to interpose functionality on objects, the facility must be transparent so that neither objects that are interposed on, nor callers of those objects need to know about or run different code paths because of any interposition taking place.

The facility must be inexpensive to use. Overhead can come from interposing and removing Arbiters, and from the overhead that is added to each call to an interposed function. The more expensive these overheads are, the more limited the usefulness of the facility.

There are also software engineering requirements. It must be relatively easy to write Arbiters. In the context of K42, this means that the interposition facility must integrate well with the K42 Clustered Object System. It also means that the Arbiters should not require more specific knowledge of the target objects than is necessary for the function of an Arbiter. For example, an Arbiter to time a function call should not need manual specialization for each type of object it can time calls for.

Finally, the interposing facility must be general and flexible. It should have as much coverage in the operating system as is practical, and should not be limited in the tasks it can perform when interposing.

1.5 Structure of Dissertation

The remainder of this dissertation is structured as follows. Chapter 2 first describes previous work related to code interposition. Chapter 3 then gives background on K42. Chapter 4 describes the design and implementation of an interposition facility for K42. In Chapter 5, the implementation is evaluated using micro-benchmarks, and some simple applications for debugging and performance monitoring are presented. The dissertation concludes with a discussion of future work, and of possible applications of code interposition as a starting point for higher level operating system features.

Chapter 2

Related Work

A number of systems have used various techniques to interpose code. Static instrumentation is a precursor to interposition, where source code is inserted at strategic points in the code before compile time. Chaining of function calls or handlers is a form of code interposition used in some systems. Other systems have used the natural boundaries within the system to allow interposing on calls that cross boundaries, such as modules or system calls. A more sophisticated technique involves replacing an instruction with a "trampoline" that transfers control elsewhere to interpose code. Some systems also use runtime systems to support things such as object orientation or modules, and these systems can also provide a means for interposing. Each of these approaches to code interposition is discussed in this chapter.

2.1 Precursor to Interposing

Although interposing occurs, by definition, dynamically, the same objectives that interposition addresses can also often be approached statically. This is the case, for example, for instrumentation. Instrumentation is inserted dynamically if the need to insert it is not anticipated, or for flexibility or cost advantages. If the need to insert instrumentation is anticipated and the type of instrumentation decided in advance, it can be added statically.

One example of static instrumentation is the common technique of inserting printf statements in C programs in order to aid debugging. These statements are added and removed as the program is being debugged, and the program is recompiled and run in order to gain insight about exactly what the program is doing. The insertion of simple counters that are incremented at various places in a program is another form of static instrumentation. This technique is used extensively in operating systems to count events like context switches or page faults suffered. Counters are used to collect many of the statistics found in the Linux /proc filesystem, which provides data about the state of the Linux kernel and user processes. Linux

has counters for memory subsystem events, network interface events, a number of per-process events, and so on.

Static instrumentation has several advantages over dynamic approaches that make it useful in some situations. First, it is relatively inexpensive for each instance of instrumentation. Often it consists of just a single increment instruction. Another advantage is the ease of adding static instrumentation. Instrumentation is inserted in the original source code. As long as the programmer makes sure that the instrumentation is correct in a logical sense, the compiler will take care of making sure that the instrumentation goes into the correct location and does not disrupt the program or cause incorrect results.

Static instrumentation also has disadvantages, however. Static instrumentation cannot be added to a running program; the program needs to be recompiled each time a change is made. Moreover, while static instrumentation is inexpensive in small amounts, in large amounts it can become expensive. This is a problem in particular because it can not be turned off when it is not needed. There is thus a lack of flexibility with static instrumentation, as needs must be anticipated at compile time because they can not be added later.

Active instrumentation, instrumentation that measures something and triggers events on some condition, is also problematic with a static approach; making the instrumentation say, "Do this, but if it happens more than twenty times use a different approach," is expensive because the check for more than twenty times must happen each time the code is executed. The check must happen even if the event is triggered and the code is run many more times, because it is compiled statically into the program. Active instrumentation using static instrumentation has a significantly higher cost, so its usefulness is limited. The lack of flexibility makes static instrumentation inferior in many scenarios.

When using languages that support dynamic recompilation, static instrumentation can be interposed in a more versatile way. Many Java Virtual Machines are able to compile code on a just-in-time basis, and some can recompile code that has been previously compiled [15]. When the code is recompiled, static instrumentation can be added or changed [3]. This instrumentation can be used to identify locations for optimization, such as code that runs often and should be compiled aggressively. Although the instrumentation is compiled in statically, it can be removed when it is no longer needed by recompiling the code. JVMs can also insert active static instrumentation to ensure correctness of an optimization. Code may be optimized by assuming that some variable always maintains the same value. Static checks can be compiled in to make sure that this is true, and to run different code if the value changes. The ability to interpose instrumentation dynamically in Java programs helps JVMs achieve better performance than they could otherwise.

2.2 Trampolines

Trampolines are jump points that cause execution to leave the current execution flow and execute interposed code instead. Trampolines are often implemented by replacing an existing instruction with a jump instruction to the interposed code. The insertion point is called a trampoline, because when the program reaches the trampoline, control jumps to the interposed code. Trampolines are versatile because they can replace almost any instruction in the program, and so can be inserted almost anywhere in a program. Since trampolines are inserted at runtime, they can be used when static instrumentation is not suitable. There are a number of systems that use trampolines to interpose code; the list presented later in this section is representative rather than complete.

Trampolines pose a number of implementation challenges. A first challenge is locating instructions to replace. Variables and constants are often mingled with program code, and replacing these with a trampoline will not have the desired effect. On x86 machines, instructions are variable length, and can be particularly difficult to locate. Replaced instructions have to be executed or emulated by the interposed code. This is difficult and is not possible for some instructions, such as PC relative jumps, under some emulation or execution schemes. Another challenge is retaining the context of the calling program. Program registers and processor status bits must be saved so that they are not overwritten by the interposed code. Another challenge is shared with static instrumentation: trampolines interpose on global code paths. This is not always a disadvantage, but it means that all threads and all object instances that run code with a trampoline will jump to the interposed code. Sometimes it is desirable to interpose for only a particular thread or object instance. To do this with trampolines requires explicit checks for thread or object instance.

The application of trampolines in a number of systems is described next.

2.2.1 Debuggers

Traditional debuggers use trampolines to break out of the normal flow of control of a program [13]. Debuggers interpose on programs that are being debugged in order to examine the program state by inspecting variables and memory locations, or to pass control to an operator that can inspect the program. Debuggers typically overwrite an instruction in the program with a trampoline instruction that raises an exception. The debugger intercepts the exception when it occurs and transfers control to the debugger. Debuggers can then perform actions such as examining and setting variables in the program being debugged, setting additional breakpoints, and single stepping through the program to allow an operator to examine execution. When the

debugger returns to the program, it uses the single step ability to execute the instruction that it overwrote. It puts the original instruction back in the place that it overwrote with its trampoline and then executes the instruction in single step mode. Flow control returns to the debugger after executing that single instruction. The debugger then puts the trampoline back in and returns from the exception. The program then resumes execution at the point after the trampoline instruction.

The approach taken by debuggers is versatile because it can interpose anywhere in a program. There are limitations, however. Generating an exception is an expensive operation, so debuggers have a high cost relative to many other systems for interposing. Debuggers have proven very valuable for debugging, but, because of the overhead incurred, are not commonly used for more general applications of code interposition, such as performance monitoring or changing program functionality.

2.2.2 DynInst

DynInst is a dynamic instrumentation library that is intended to add instrumentation to a program to monitor its behavior [8]. DynInst provides a simple instruction set that can be used to write simple instrumentation programs. DynInst is also able to call functions in the program it is interposing on. This makes DynInst capable of more complex modifications to a program's functionality, since it can use the capabilities of the original program to modify data. For example, a programmer could anticipate the use of DynInst to interpose code to change a policy and provide functions implementing different policies. DynInst could then be used to interpose on a location where a decision is made, and call a function to make the decision using an alternate policy. DynInst uses trampolines to dynamically insert instrumentation or functionality at runtime. When an application is running, DynInst can send a signal to interrupt it and overwrite an instruction with a trampoline to jump out to interposed code, similar to the way that a debugger works.

In order to ensure correctness, DynInst has a sequence of steps that it goes through to run interposed code. The trampoline jumps to block of code that consists of (i) a jump instruction leading to a pre-instruction instrumentation block, (ii) the overwritten instruction, (iii) a jump to a post-instruction instrumentation block, and (iv) a jump back to the original code. This block of code allows instrumentation blocks to be run before and after the instruction that has been interposed on, and it solves the problem of how to execute the instruction that has been displaced by the trampoline. To execute the interposed code transparently, DynInst saves the registers and other necessary machine state such as condition code registers in each instrumentation block and runs the instrumentation code that has been installed. As part of the process,

DynInst checks the current thread in order to ensure that instrumentation only runs on the thread for which it is intended. It then restores the registers and machine state and jumps back to the program code in which the trampoline was inserted.

This method does not work with many instructions that affect control flow, such as jumps and returns, so instrumentation blocks can only be added around some instruction types. To decide where to install a trampoline, DynInst needs to examine images of programs that include a symbol table. This allows DynInst to find function entries. DynInst then parses the program to find other valid insertion points. Points that DynInst identifies as suitable for inserting instrumentation are the beginning and end of functions and the beginning of all basic blocks. DynInst also allows the operator to specify an address to insert a trampoline at. This has limitations though, since the operator must decide manually which sites are valid, and the program may execute improperly or crash if the operator puts a trampoline in a bad location.

2.2.3 KITrace

KITrace, the Kernel Interactive Trace tool, dynamically inserts instrumentation code into operating system kernels using trampolines [16]. KITrace runs on various types of hardware, including Motorola 68000 and Sun SPARC systems running SunOS and Intel 80386 systems running Linux. KITrace uses a small component inside the operating system kernel that includes code to insert trampolines and instrumentation code, and space for a trace log. KITrace also has a user space component that commands the kernel component and reads and interprets the trace log.

KITrace requires some facility to allow insertion of code into the operating system kernel. SunOS has a standard facility for inserting code into the kernel, and a patch to the Linux kernel allows KITrace to do this on x86 Linux machines. To ensure safety, KITrace allows only a limited set of instrumentation commands that can be used to write instrumentation programs. KITrace does not support running of user supplied C functions and it does not allow the calling of functions within the kernel. KITrace instrumentation commands specify what data should be collected in the KITrace trace log for observation and analysis at a later time. KITrace trampolines are not implemented with jump instructions, but with trap instructions, so there is a relatively high overhead each time a trampoline is executed.

Like debuggers, KITrace trampolines save processor state and set up a an execution environment. When the interposed instrumentation returns, KITrace executes the overwritten instruction in single step mode, before returning to the original program. By doing this, KITrace is able to interpose safely on a wide range of instructions, including instructions such as branch and return, that change the location of execution, and

thus is able to interpose in more different places than DynInst.

2.2.4 DTrace

Another interactive tool that uses code interposition to generate traces of operating system events is Sun's DTrace tool for Solaris 10 [9]. DTrace has both user space and kernel components and is able to interpose instrumentation in both user space and kernel portions of the operating system. DTrace uses trampolines to interpose in a way similar to DynInst. The feature that DTrace offers over other trampoline based interposing frameworks is safety; DTrace has been designed so that unprivileged users can install instrumentation in the kernel with no risk of crashing the operating system or causing incorrect behaviour, either of which could happen if systems such as KITrace were used improperly.

DTrace uses these techniques to ensure that unprivileged users interposing code in the operating system kernel will not affect the integrity of the operating system. First, instrumentation can only be inserted at predetermined locations in the code. These locations include all function entry and exit points, as well as a large number of statically predetermined locations. Secondly, instrumentation is written in an application-specific scripting language that does not allow loops and can be statically checked for safety properties. Thirdly, when instrumentation code is being executed, interrupts are disabled and any exceptions generated go to an instrumentation handler that (i) records the fault that the instrumentation generated, (ii) stops execution of the instrumentation and (iii) carries on with the original program.

To install instrumentation, scripts, written in the DTrace scripting language, are submitted to the DTrace subsystem for validity checking, along with a location specifying where to interpose. Together, these are called a probe. DTrace maintains a list of probes that have been checked. For a program to use instrumentation, it specifies a probe that it wishes to install, and DTrace installs the probe at the request of the program by creating a trampoline and overwriting the instruction at the location to be interposed on.

Trampolines in DTrace have several different implementations depending on the location being interposed on and the machine architecture. There are two distinct kinds of locations: function boundaries and statically defined locations. Statically defined locations are identified by annotation in the program source code. The compiler inserts a no-op instruction in the object code at the specified location. To interpose instrumentation, the no-op instruction is replaced with a jump into a trampoline that calls the instrumentation code. Using this strategy, the replaced instruction (i.e. the no-op) need not be executed. The trampoline saves and restores machine state before executing the target script.

To insert instrumentation at function boundaries, instructions at the beginning or end of the function are

replaced with a jump instruction for SPARC or a trap for x86. The jump instruction on SPARC machines leads to trampoline code that saves machine state and then jumps to a function to run the instrumentation script. Upon return, machine state is restored and the original instruction is executed within the trampoline code before returning control to the original code. The trap on x86 is handled by a special handler that sets up the machine to run the instrumentation script, looks up the correct instrumentation script based on the instruction pointer of the trap, and passes control to the function to run the instrumentation script. Upon return, the replaced instruction is emulated. The original state is then restored and the trap returns.

2.3 Interposing at Logical Boundaries in Systems

Some systems support code interposition at natural boundaries of the system. Since boundary crossings often represent some logical divisions and have clear interfaces or conventions, it is often easier to interpose at these interfaces. Code interposition then allows one to monitor or modify interactions between the parts of the program. Trap handlers and jump tables are examples of places that allow for relatively simple interposition by replacing one address with a different one, as with the PC-DOS interrupt and system call handlers.

2.3.1 PC-DOS Interrupt Service Routines

PC-DOS supports the interposition of interrupt handlers and system call handlers dynamically at runtime [12]. In PC-DOS, all interrupts and system calls are routed through an interrupt vector table located at a fixed location in memory. The system calls and interrupts form a boundary on which code can be interposed. An interrupt or system call handler to be installed is interposed by first reading and locally storing the entry at the corresponding location in the vector table and then overwriting the entry to point to the code being interposed, so that subsequent interrupts or system calls result in the interposed code being executed. The interposed code can then explicitly call the previous handler, if necessary, by jumping to the code originally pointed to in the vector table entry. Thus, the handlers can be chained so that all handlers associated with the interrupt or system call are called. Effectively, new handlers are interposed in front of previously installed handlers.

2.3.2 The Mach Operating System

The Mach microkernel [1] supports code interposition at the system call boundary by having the ability to reflect system calls back to the user level process that generated the system call [14]. Call interposition

is used to enable Mach to run UNIX and MS-DOS binaries. User level libraries and servers handle the functionality that would traditionally be in the UNIX kernel. This is in line with the Mach project goal of providing a microkernel that is suitable for constructing a modern UNIX like operating system.

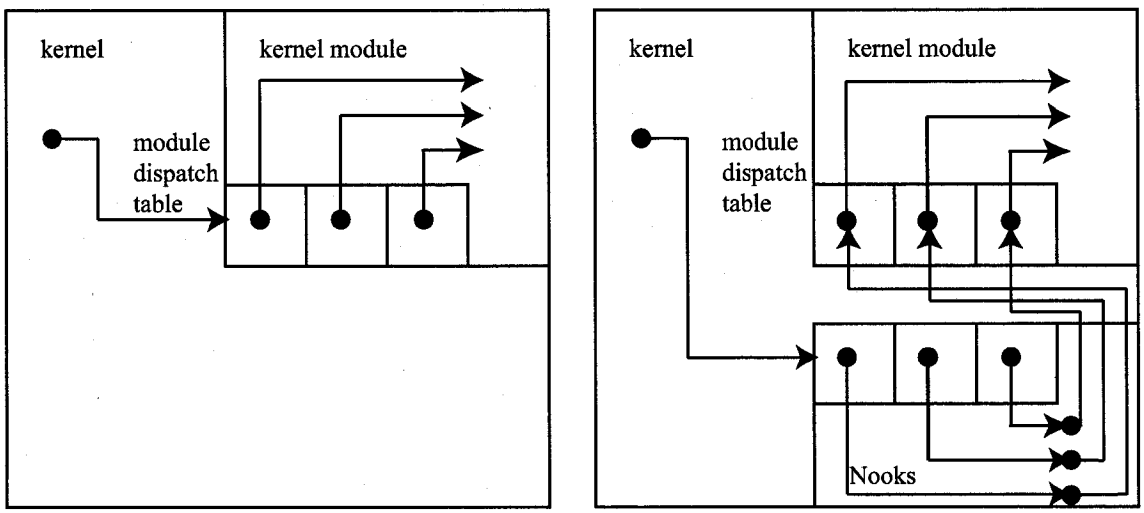
Mach does not provide a complete UNIX environment within the kernel; instead, a user level library within the address space of UNIX applications provides the missing services and interfaces that are compatible with the particular UNIX version for which the application was compiled. UNIX applications that dynamically link to libc are instead linked to the Mach user level library. System calls that libc would normally make are instead handled internally by the library, or by making appropriate calls to Mach servers and kernel as necessary. For applications that make system calls using a mechanism other than a dynamically linked libc, such as a statically linked libc that make system calls directly, this library replacement is not sufficient. To allow the Mach user level library to handle the system call even in such cases, Mach interposes on the system call boundary and reflects calls that are not made by the Mach user level libraries back to the calling process so that the Mach libraries can handle the system call.

Mach has a function that instructs the kernel to intercept system calls based on the type of system call and the address space it comes from. The Mach interposing toolkit uses this function to redirect to itself any system calls that it wants to interpose on. When it receives an intercepted system call, the interposing toolkit examines the list of handlers it has registered and decides where to redirect the system call to. This can include user level code in the address space that made the system call. When the handler finishes dealing with the system call, it restores the previous state and returns to the application. This mechanism is not limited to implementing operating system features that are not provided by the Mach microkernel; it can also be used to execute arbitrary user level code on a system call.

2.3.3 Linux Kernel Modules

The boundary between the Linux kernel [17] and loadable Linux kernel modules is another place where code interposition is possible. Many Linux kernel components can be either linked statically into the kernel or compiled as relocatable object files and linked dynamically at runtime. Some third party kernel components are available as loadable modules that can be linked into the system dynamically at run time. Loadable kernel modules can be filesystems, protocols or other system components that are not necessary for the basic functionality of the kernel. Device drivers are the most common type of kernel module.

Users of the system with appropriate permissions can load kernel modules at run time, or loading can be done automatically in order to satisfy requests by programs or other kernel components, if the appropriate



Linux kernel

Linux kernel with Nooks reliability subsystem

Figure 2.1: Linux Kernel Modules and Nooks Reliability Subsystem

Linux kernel infrastructure is enabled. Since the modules are loaded dynamically, it is not possible to know in advance where in memory they may be placed. Instead, a dispatch table in the module contains the relative position for all of the functions that the module exports for the rest of the kernel to use. To access a function in the module, the kernel uses a pointer to the dispatch table and an index number for the function that indexes into the table to get the address of the function it wants to call. This level of indirection is what enables Linux kernel modules to work, and is shown in the left side of figure 2.1.

The dispatch table and the indirection through it form a boundary between the kernel module and the rest of the Linux kernel that can be easily interposed on. To interpose on the functions of a kernel module, it is only necessary to maintain a shadow dispatch table that points to its own functions instead of the functions in the module, and set the pointer that normally points to the dispatch table to point to the shadow table instead. Interposing functions can call the original functions in the kernel module by caching the function's address, or by using the original dispatch table located in the module.

As an example, the Nooks reliability subsystem for Linux uses shadow dispatch tables to interpose on calls made to Linux kernel modules [22]. The general approach as used by the Nooks system is shown in the right side of figure 2.1.

Nooks attempts to protect the kernel from faults in device drivers and other kernel extensions through

two mechanisms. First, whenever a function in an interposed module is called, and when it returns, Nooks checks the parameters to make sure they are valid. This includes pointer validity checks and possibly other checks specific to a particular module. The second thing Nooks does is isolate the kernel from the modules. Modules interposed on by Nooks are run with reduced permissions so that they can not overwrite memory belonging to the kernel. Data structures used by modules are copied back into the kernel by the interposed code. Nooks tracks changes so that they can be checked for correctness before being copied back into the kernel, and also so that the changes can be rolled back if the kernel module crashes.

2.4 Extensible Operating Systems

An extensible operating system is one that allows code to be downloaded into the kernel by user level programs. Extensions can be used to modify policies or implement new services or support new hardware devices. There are two significant challenges when designing extensible operating systems. The first is enabling extension code to be loaded into the kernel and dispatched. The second challenge is ensuring that extension code loaded by user programs does not compromise the integrity of the kernel.

2.4.1 SPIN

The SPIN operating system was designed to be extensible by using code interposition within the kernel [19]. In SPIN, all function calls are treated as events, and the functions are called as event handlers. SPIN uses a runtime system that allows event handlers to be replaced or interposed on dynamically, and SPIN supports the downloading of event handler functions at runtime. This allows SPIN to be extended to do things that it can not do in its basic configuration. For example, SPIN can be extended to emulate a MACH system call interface.

Extensions consist of one or more modules that are loaded into the system dynamically. A module is made of an interface with one or more functions, each of which handles some type of event. When SPIN extensions are loaded, the dynamic linker resolves references in the extension to code in the system, then runs the module's initialization code that register handlers for events so that they will be called when appropriate. SPIN also registers any new event handlers provided by the extension so that it can be called by other modules within the system.

SPIN is written in Modula-3 and takes advantage of the Modula-3 runtime to interpose event handlers. Interfaces for event handlers are found in the interfaces of SPIN modules. New interfaces are passed to the

kernel when a new module is loaded. The runtime allows handlers to look like and be called as function calls; however, there is a dispatch system that allows for more sophisticated treatment than a simple function call. When an event is raised with just one handler attached, the runtime system dispatches the single handler as a function call. If multiple handlers are attached, then the runtime system dispatches each of them.

Extensions can also have guard functions that get installed together with each of their handlers. Guards determine if the associated handler should run in a particular invocation. Guard functions are functions that have no side effects and return a boolean value indicating that the event handler they guard should or should not be executed on this invocation of the event. Often, guard functions are used to ensure that a handler is executed only for a particular application, but there are other uses as well.

The runtime system gives good safety properties for the execution of downloaded handlers. Handlers are procedures and hence have a type signature. They can only be installed for interfaces that have the same type signature. This limits the scope of event handlers and makes instrumentation harder to write, but it means that handlers can not be installed in places where they might be incorrect or dangerous. Type compatibility will not catch all possible mismatches between handlers, however, since different interfaces may have the same type signature.

2.4.2 VINO

The VINO operating system [21] is also designed for extensibility by supporting interposition of functions in the kernel. VINO is particularly concerned with allowing policy and priority decisions to be replaced by users. VINO kernel extensions are written in C++ and can be interposed at predetermined locations. VINO maintains a registry of objects and functions that can be extended. VINO is designed so that policy decisions are made in separate objects, and functions in these objects are registered as extendable. Functions for computation and stream handling are also made extensible. At runtime, extensions can be loaded and later removed, and extendable functions can be added to or removed from the registry. VINO uses sandboxing and static analysis to ensure that extensions will not compromise the integrity or forward progress of the system.

The VINO kernel maintains registries of functions both globally and per-process. Per-process functions can be extended by the process, but global functions can only be extended by the process that created the object the function belongs to. Entries in the registry consist of link information needed to interpose on the function, and a name. VINO has system calls so that user processes can look up entries in the registry by

name, load and interpose new extension code, or remove and unload an extension.

VINO extensions are statically checked and run in a sandbox. This is considered acceptable from a performance perspective, because usually only small amounts of code are needed to make policy and priority decisions. Running in a sandbox that protects memory belonging to other parts of the kernel incurs some performance penalty; however the overhead is less than that of the context switch that would be necessary if the extension code was run in user space. The static checker examines the assembly code for the extension, and does not need the original source code. The static checker generates a stub function for the extension function that sets up the sandbox; it creates a new stack for the extension, sets up memory protection and initializes regions for the extension to read and write, before calling the extension function.

To interpose an extension, the kernel first loads the extension code and then examines it before interposition. The examination verifies that the static checker has signed the extension, and determines what functions in the VINO kernel the extension uses, and ensures that it has permission to do so. Extensible functions are called indirectly, so interposing is done by replacing the pointer to the original function with a pointer to the stub function that calls the extension. After this is done, calls to the extensible function will be intercepted by the extension. Dismantling of the sandbox for the extension function is also done by the stub function before returning control to the caller.

2.4.3 K42

Code interposition already exists in the K42 operating system in two places. First, the K42 object system uses code interposition to instantiate K42 clustered object instances that have not been accessed before. Second, the existing hot-swapping infrastructure in K42 uses code interposition to block calls until an object can be safely swapped. Both of these interpose using similar methods.

The K42 object system uses an indirection table to locate K42 clustered objects. Code is interposed in K42 by changing an entry in this table to point to an interposed object. A "default object" is initially interposed on all K42 clustered objects; its job is to locate an object instance to handle a call, and instantiate a new instance if one does not exist yet. This is explained fully in chapter 3. The default object saves the program state that is needed to make the function call, sets up a C execution environment, and then calls a function to locate an object instance. After an object instance is located, the state is restored and the function call is made to that instance.

To allow hot-swapping in K42 a mediator object is interposed in front of the object to be swapped. Like the default object, the mediator saves the program state that is needed to make the function call, sets up a C

execution environment, and then calls a function to block the original function call until it is safe to proceed. After the target object has been replaced by a different instance (i.e. swapped) the mediator unblocks the calls by restoring the call path's state and making the function calls. Unlike the default object, the mediator may have to perform additional work after the original function call has been made. Only minimal program state needs to be saved, but some information does need to be passed between the pre- and post-function mediator calls. In a normal program, this information would be stored on the stack, but the mediator is not able to use the stack. A mediator table of contents is required. Since the table of contents is globally the same, it is stored in the program text and its location is retrieved by calculating the address relative to the program counter. The other information that the mediator needs to store is a pointer to the mediator object. This is different for each mediator instance, so thread-specific storage is needed. The mediator stores the pointer in a non-volatile register, and stores the original contents of the register in a hash table.

The work presented in this dissertation differs by offering a more general solution to interposition that allows arbitrary code to be run before or after the original function call is made, or allows the function call to be aborted. In addition, the local storage problem is taken care of by allowing the stack to be used by interposed code.

2.5 Summary of Interposing Techniques

Examination of related work shows that there are two common methods for implementing code interposition. The first method is to use trampolines to jump out of an instruction stream. The second method is to use an existing level of indirection.

Trampolines are implemented by overwriting some instruction in the program with an instruction that causes the program flow to be modified, allowing code to be interposed. The interposed code may examine or change the state of the program it is interposed on. Figure 2.2 shows an instruction stream, with each box representing an instruction. The second part of the figure shows an instruction stream with a trampoline in place of the load instruction. The trampoline jumps to the interposed code. When the interposed code completes, it executes the original instruction and then jumps back to the original instruction stream immediately following the trampoline. As the previous examples have shown, there are multiple ways to jump out of the original instruction stream, and multiple ways of executing the code that has been overwritten, however the same general approach is taken in each case.

Trampolines are versatile because they can interpose almost anywhere in a program. Although not all

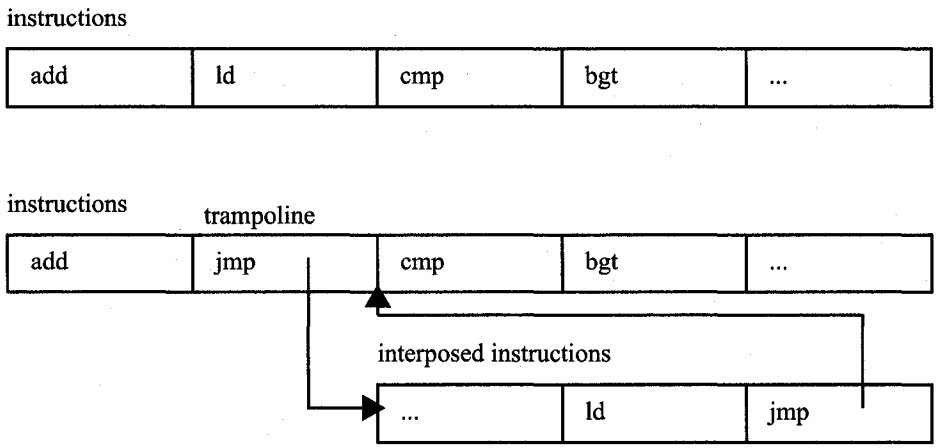


Figure 2.2: Interposing Using a Trampoline

methods can insert a trampoline at any instruction, they are all able to choose from a very wide variety of locations. Trampolines do not need to have an interface or boundary to interpose on.

Using trampolines for interposition also presents some challenges. Trampolines are usually specified in terms of a particular instruction to interpose on, which requires parsing the object code at a low level. If some instructions can not be interposed on, they must be identified by the interposing system. Trampolines expose the interposed code to a broad part of the program's state, so transparency is a challenge. It may be necessary to save all machine registers and to be careful not to overwrite any part of the program's stack. Another challenge is executing the instruction that was overwritten by the trampoline. There are several approaches to this, but no single approach seems to be best for every case. For example, in some cases it is necessary to emulate the replaced instruction instead of just executing it.

The second common form of interposition exploits a level of indirection that already exists in the system. An indirection table is often used to locate functions whose final location or value is not known at compile time. The indirection table shown in figure 2.3 is a virtual function table. In the first part of the figure, the table has pointers to various functions that a system uses. In the second part of the figure, the first and third functions have been interposed on by replacing the pointers in the indirection table. The new pointers go to the interposed code, which can examine and modify the system state, or block, redirect or delay the function call.

Interposing through an indirection table is convenient because many systems already have indirection tables, so the applicability is reasonably broad. Indirection tables usually exist at a function granularity. They are not as versatile as trampolines that can be interposed almost anywhere in a program, but they are

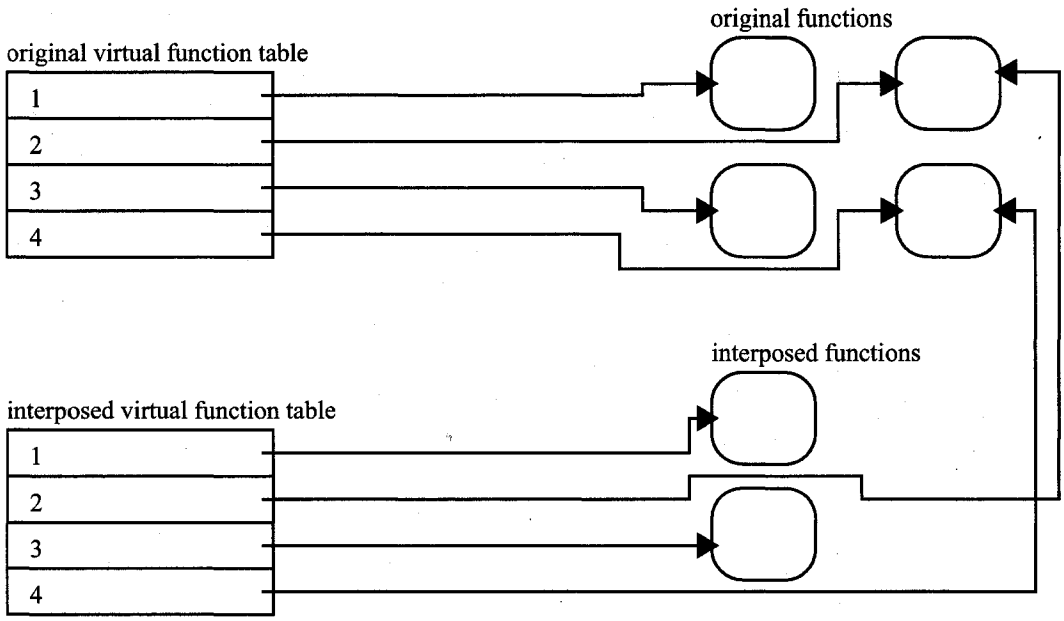


Figure 2.3: Interposing Using an Indirection Table

often good enough for monitoring or modifying specific functionality in a system. Indirection tables also require less knowledge about the specific code being interposed on, because indirection tables provide a consistent interface. In contrast, trampolines are not valid for all instructions, and so the code being interposed on must be parsed more carefully to determine which ones are safe to interpose on. In some ways, indirection tables allow for more versatile interposition than trampolines. Indirection tables make it much easier to replace code that is interposed on with different code. With an indirection table, it is merely necessary to run different code in its place. Using trampolines, this would require knowledge of the target program to know where execution could safely continue, and what patch up work would be necessary to jump execution to that point. Due to the difficulty involved in this, trampoline based systems usually require that the instruction displaced by the trampoline be executed, making it possible to add interposed code, but not to replace existing code.

The primary disadvantage of interposing through indirection tables is that it can only be done in a system that contains indirection tables. Maintaining transparency is also challenging when interposing through indirection tables. Although the indirection tables provide a consistent interface, the calls that go through these tables often involve transferring significant amounts of state between the caller and the callee. This state that must be saved and later restored so the interposed code can run without being noticed by the caller, or the function that is interposed on, if it is called by the interposed call.

The system proposed in this dissertation uses indirection tables for interposition. The described work was done in the context of the K42 operating system which contains indirection tables that are used for invoking functions for all objects in K42.

Chapter 3

The K42 Operating System

K42 is an operating system that is designed to be flexible, adaptable, and efficiently scalable to large multiprocessor shared memory computers. It is currently being developed by IBM Research in collaboration with the University of Toronto and other academic institutions. K42 is heavily influenced by the Tornado operating system developed at the University of Toronto [10].

Achieving good performance for shared memory multiprocessor programs has received considerable attention. K42's overall structure, algorithms and data structures have been designed to achieve good multiprocessor performance without sacrificing uniprocessor performance. The K42 philosophy is that scalability is best achieved by minimizing shared data and global code paths. This is accomplished with an object-oriented approach, where every virtual and physical resource in the system is represented by an independent object instance to ensure natural locality and independence for all resources. Locks are internal to the objects that they protect, and global locks are not used.

Parallel applications have many specialized needs, from specific memory layout to particular communication and scheduling demands. Here, K42's adaptability 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 requirements of supporting the system across different architectures (PowerPC, EM64T, 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 is conditionally compiled using preprocessor directives, 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 adaptability in a first class way. A

running K42 operating system can use resource implementations chosen at run time specifically for the workload it is running and the hardware it is running on. It can change the implementation of any object at run time to adapt to changes in workload. This way, K42 achieves the benefits associated with other customizable operating systems that can tune for application behaviour [4]. K42's object-oriented approach also yields significant benefits in terms of structuring and programmability of the operating system.

K42 achieves scalability and adaptability using object oriented structure along with K42 clustered objects. The object oriented structure of K42 allows scalability by minimizing data sharing and avoiding global locks, and allows adaptability by defining interfaces that can be implemented in different ways and composed to allow different configurations. K42 clustered objects are special objects that have a structure that allows for locality optimizations for objects that are inherently stared in a multiprocessor environment. K42's object oriented structure and clustered objects are described in more detail in the remainder of this chapter.

3.1 K42's Object Oriented Structure

K42 uses an object oriented structure. Each resource or service is represented by object instances. Access to a resource or service is obtained by making a function call to the object instance representing the resource. This allows K42 to contain multiple implementations of an object for a service; each one specialized for some particular usage scenario. K42 then adapts to the running application by instantiating objects optimized for the needs of the application. For example, objects that read files from disk may be optimized for small file sizes.

Another performance benefit from object orientation is the logical separation of different components, with attendant reduction in sharing. This gives performance benefits because locks and data are not shared by unrelated parts of the system. Since different instances of services, such as different files on disk, are accessed through different object instances, their data is logically separated and they can be accessed simultaneously on different processors without sharing. Figure 3.1 shows how the object oriented structure of K42 reduces implicit sharing. On the left side is a traditionally structured operating system. The system shares global code paths and data structures. Even if the locking granularity is reduced, many things are shared implicitly. The right side of the figure shows an object oriented structure. Since different code paths use different data structures belonging to different objects, sharing is reduced implicitly. K42 clustered objects extend this benefit further by making it convenient to partition objects representing resources so that

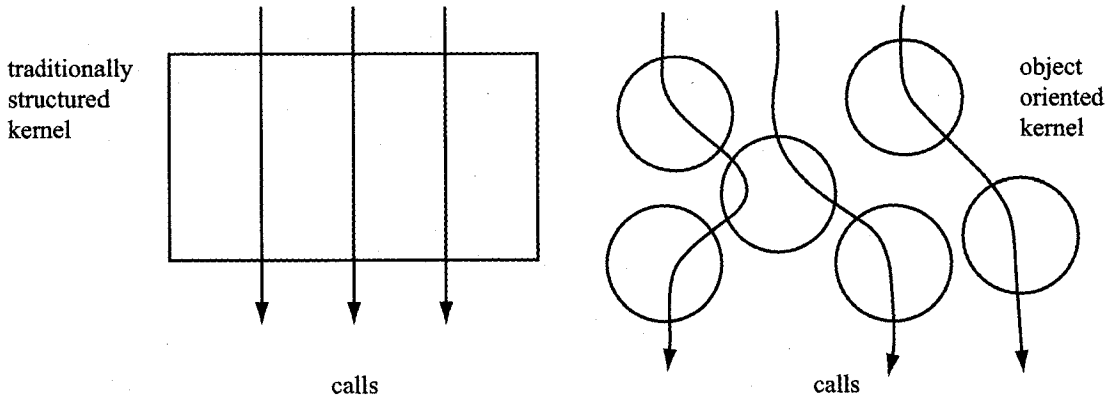


Figure 3.1: Traditional and Object Oriented Operating System Structure

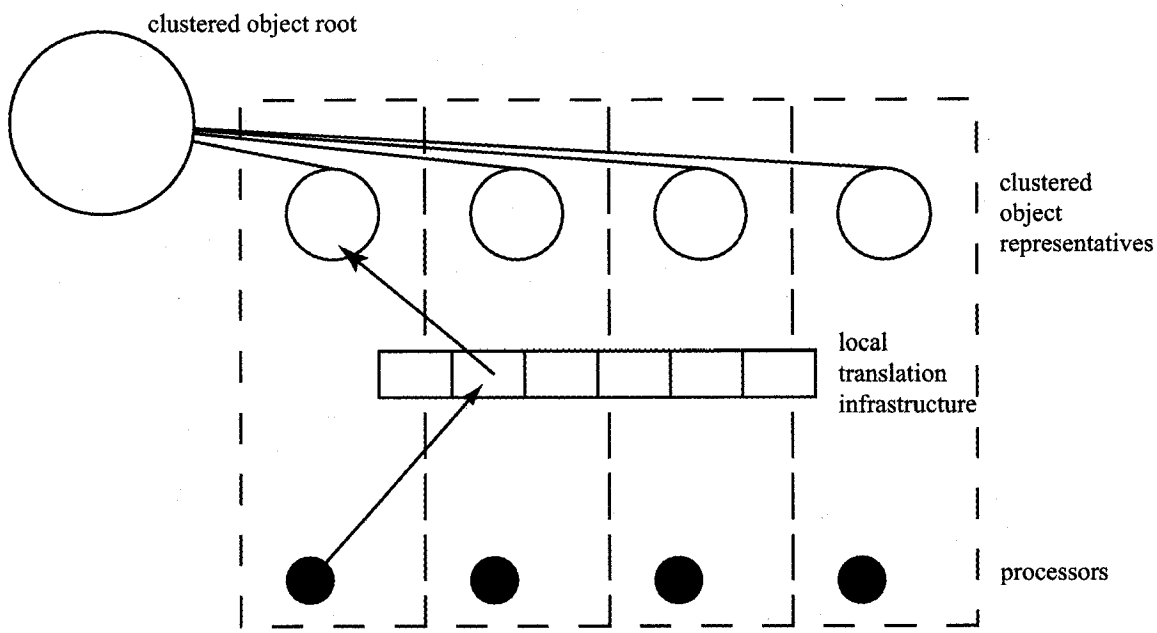


Figure 3.2: Logical View of Clustered Objects

requests from each processor can be handled independently.

3.2 Clustered Object Overview

K42's object oriented structure alone does not necessarily achieve good multiprocessor performance, since some objects end up being shared. For example, in the K42 virtual memory manager, the address

space mapping list in the process object and the free page list in the global page manager object can became bottlenecks in a large multiprocessor system because of shared data access from multiple processors. For these objects, locality needs to be maximized to avoid slower remote memory accesses, minimize lock contention, and reduce cache-coherence traffic. Clustered object technology extends the object-oriented design by providing management of 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 [2].

From a client's perspective, a clustered object appears similar to a C++ object; that is, their interfaces are the accessed and used through similar mechanisms. A logical view of clustered objects is shown in figure 3.2. A clustered object is logically a single object, but internally it is composed of one or more component objects called representatives. Each representative handles calls from a specified subset of the processors. A clustered object is accessed via a clustered object identifier, which is common to all processors. Function calls on clustered objects are done using this identifier. The local translation infrastructure automatically redirects each call to the appropriate representative based on the processor from which the call is made. A representative may be responsible for calls on any number of processors as decided by the clustered object, but most clustered objects have either one representative handling calls from every processor, or one representative per processor (as shown in the figure). Each representative provides its clients with the full functionality of the clustered object. If necessary, the representatives of the clustered object communicate with each other to maintain consistency.

In addition to the representatives, each clustered object has a root object. The clustered object root is responsible for maintaining the list of representatives, and creating new ones as needed. The root may also serve as a place to store information that is shared by all of the representatives.

The internal data representation and algorithms of the clustered object are transparent to the client. If the shared object data is read mostly, replication may be adopted, with each processor's local representative maintaining its own replicated copy of the data. Some objects are partitioned so that the data most accessed by a processor will stay local to that processor. With appropriate internal implementation, an object can be optimized for locality and concurrency depending upon an assumed access pattern. With implementations involving multiple representatives, it is necessary to keep them consistent. While 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 client remains the same. While the internal data may be replicated, migrated, or partitioned, the clients can make function calls to the object without knowledge of its actual

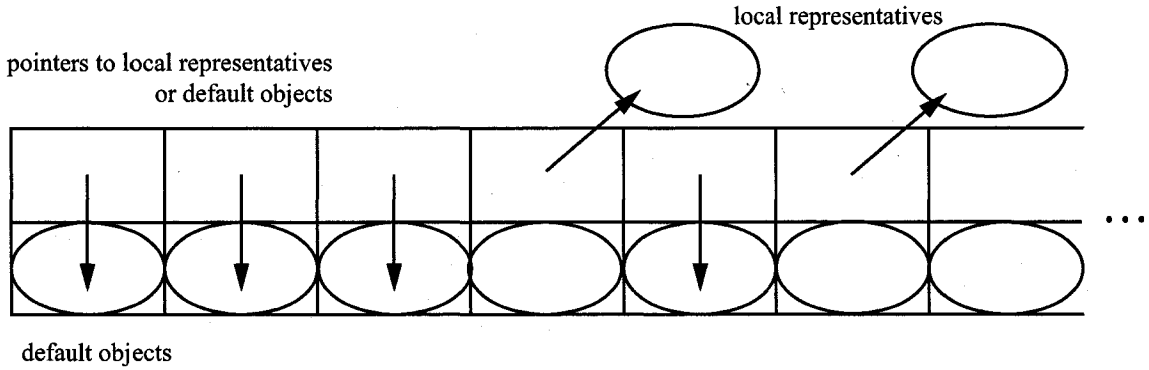


Figure 3.3: Sample Local Translation Table

implementation.

The clustered object infrastructure of 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 representatives 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-representative clustered object whose implementation would be almost identical to that of a common non-clustered object. If the object becomes a bottleneck, a multi-representative clustered object could be implemented and used instead. Since the interface remains consistent, implementations with different degrees of clustering and consistency protocols can be experimented, without modifying the rest of the system. The interface provides the flexibility to allow different implementations of a clustered object to exist, each of them optimized for different usage requirements.

3.3 Implementation of Clustered Objects

In order to implement clustered objects, more infrastructure is required than C++ provides. K42 clustered objects require an extra runtime system that allows clustered objects to instantiate representative objects and locate an appropriate representative on calls to clustered object functions. C++ provides several features in its runtime system, including runtime type identification, exceptions, virtual functions and multiple inheritance. Use of some features of the standard C++ runtime is restricted within K42. In particular, K42 does not permit exceptions, runtime type identification or multiple inheritance. These features cannot be supported alongside the clustered object runtime without adding significant extra complexity. The K42 runtime system and the K42 interposing framework do, however, make extensive use of virtual functions and their underlying implementation in C++.

3.3.1 Local Translation Tables

Clients access a clustered object by means of a clustered object identifier. The identifier is actually a pointer to an entry in a per-processor table, called a *local translation table*, which contains pointers to the corresponding local representative objects. When a representative has not yet been installed for a clustered object, then the entry points to a stand-in object called the *default object*. The default object allows the translation mechanism to determine the correct representative if the pointer to the local representative has not yet been set. Figure 3.3 depicts a local translation table with some entries that have local representatives installed and some that point to their default objects. Each entry in the local translation table contains two things, (i) a pointer to a clustered object representative or default object and (ii) a default object. Entries for objects that do not have a local representative on the current processor point to the default object in the second part of the translation table entry.

The local translation table is defined on a per-processor basis. The entries on different processors for a particular object could point to the same representative, or to different representatives, depending upon 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, K42 uses aliased virtual memory, 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 local translation table, located at the same virtual address. Since many objects are only accessed on the processor on which they are created, ownership of the table is partitioned into disjoint sub-ranges, one per processor. This way, allocation of entries in the table does not require synchronization across processors.

Clustered objects use virtual functions, which means that calls to functions in clustered objects go through the local translation tables. This is necessary so that the default object can be called and miss handling invoked when there is no representative of the clustered object on the current processor. For normal C++ objects, virtual function calls are made by locating the object instance, then looking in the object instance to find the object's virtual function table. The target function is then located by looking in the virtual function table. The process is similar for clustered objects. The difference is that the object instance for a clustered object is the local representative that is specified by a clustered object identifier. To make a virtual function call, the local representative is found by dereferencing the clustered object identifier, which is actually a pointer into the local translation table. In the local translation table is a pointer

to a clustered object representative, which is either a default object or the desired object instance. If it is the desired object instance, then the call is made in the same way as a normal C++ virtual function call. If the pointer is to a default object, the call is made to the default object, which uses the miss handling procedure that is described below to locate a local representative and make the call.

The local translation table is a key location for interposition in K42. By placing pointers to interposed objects in the local translation tables, code can be interposed. The miss handling system is used to ensure that interposed objects are placed in the local translation tables.

3.3.2 Miss Handling

Representatives are created lazily. This is done for two reasons. First, 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, and it would be wasteful to install a representative on every processor on a multiprocessor if only a small subset of the representatives would be used. Further, lazy creation spreads out the creation time, so that there is no pause after a clustered object is created while all of the local representatives are created.

Lazy creation is accomplished by initially installing a pointer to a generic handler object called the default object, instead of to a local representative, in all local translation table entries. The purpose of the default object is to modify the table entry to point to a particular representative on demand when the processor invokes a function on the clustered object is for the first time [20]. This way, processors that do not access a particular object will not need to perform unnecessary setup. The process of setting the translation entry point to point to a representative 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, a clustered object with multiple representatives must manage the set of representatives and instantiate a new representative corresponding to the faulting processor if it has not already been instantiated. In the clustered object system, the object that manages the set of representatives is called the *miss handler*. The miss handler is responsible for object-specific miss handling, and is instantiated when the clustered object is instantiated. The pointer to the miss handler is installed in an auxiliary table called the *global translation table*, indexed the same way the local translation tables are. The structure of the global translation table is shown in figure 3.4. In addition to a pointer to the object-specific miss handler, the global translation table entry for a clustered object contains a number of status flags. The miss handler is part of the root of the

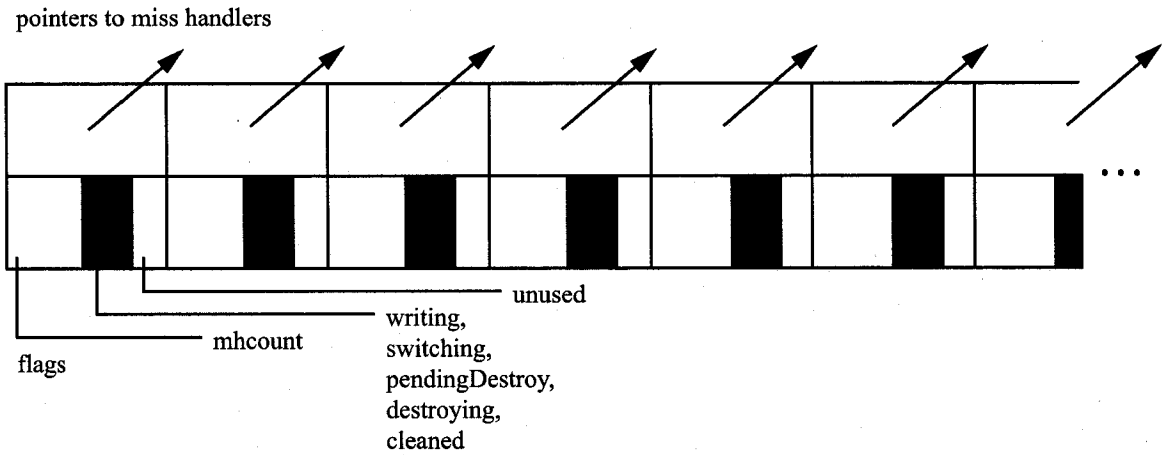


Figure 3.4: Sample Global Translation Table

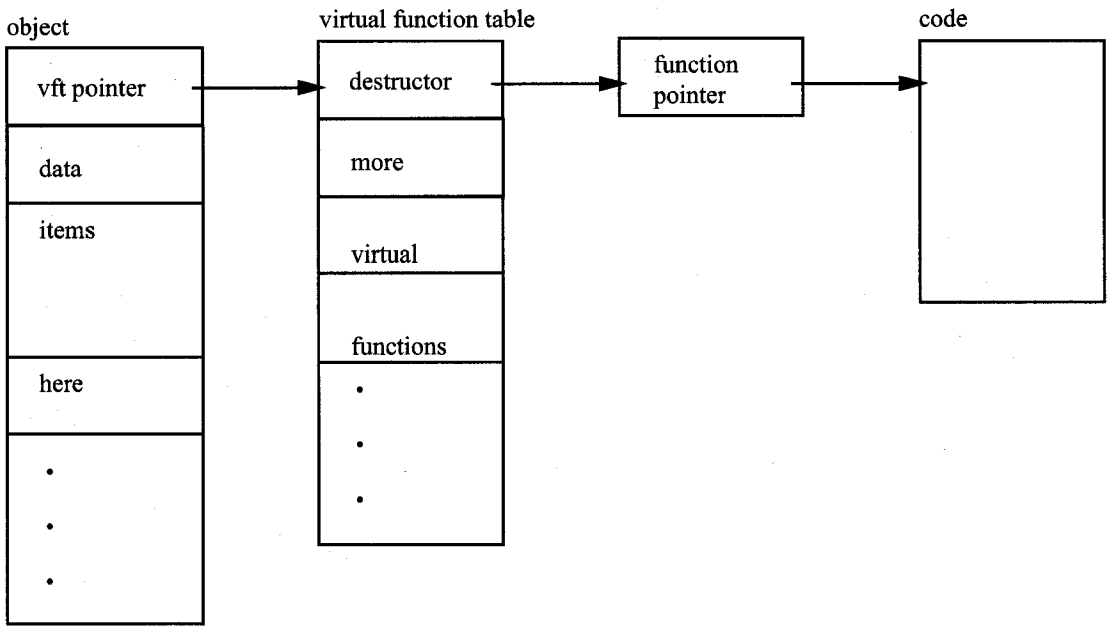


Figure 3.5: Object Layout with Virtual Function Table

clustered object.

Entries in the local translation tables are initialized to point to a generic handler object called the default object. The default object uses a special generic function that is compatible with any set of parameters to invoke the corresponding miss handler and its object-specific miss handling code. The result of invoking this miss handling code is a pointer to a representative that is responsible for handling the call. After obtaining

a pointer to the appropriate representative from the miss handler, the default object forwards the original invocation to the representative. This forwarding is transparent to both the representative and the client that faulted. Once a pointer to a representative in the local translation table has been installed during miss handling, subsequent function calls to the clustered object will be handled by the representative directly.

The implementation of the default object used to perform miss handling transparently is closely tied to the implementation of virtual functions in C++. Figure 3.5 shows the layout of a C++ object with virtual functions, as generated by the compiler for K42. The first element of an object is a pointer to the object's virtual function table, which is followed by the data members of the object. The virtual function table contains a pointer to a function descriptor for each virtual function in the class. Each descriptor contains a function pointer that points to the code for its respective functions.

The default object is allocated and assigned a virtual function table with enough generic functions to support any clustered object. Since the default object knows nothing about the invocation context, each virtual function of the default object, when called, saves all the registers of the original caller before performing the miss handling work. Once the miss handling is done and a representative is obtained, the default object restores the register contents, replaces the this pointer argument with the pointer to the representative that will handle the call, and forwards the call to the corresponding function of the representative by looking up the representative's virtual function table. While the operation has a non-negligible overhead, it is typically performed only once to establish the translation table entry, and only infrequently thereafter[1].

A consequence of using the virtual function table for miss handling and call redirection is that all clustered object functions need to be virtual. While in some ways this is a restriction, it also offers advantages in that inheritance and polymorphism can let clients access clustered objects without being concerned about what particular implementation is being used.

Miss handlers are important for code interposition because they locate the function to call when the local translation table is empty. A new miss handler must be interposed for a clustered object that is being interposed on. In addition, interposed code can be called by clearing the local translation tables and letting the interposed miss handler handle any faults that are generated.

3.3.3 Clustered Object Destruction

One final responsibility of the clustered object system is to ensure proper destruction of clustered

[1] The local translation tables are caches that can be cleared, for example if they are paged out of memory. The local translation table entries are not saved in this case since they can be re-established faster by faulting and getting the entries from the miss handlers.

objects. Since clustered objects can be used concurrently from different processes, there is a potential problem in deciding when it is safe to destroy a clustered object. K42 solves this problem by allowing objects to be submitted for destruction, but postponing the actual destruction until the object is known to be no longer in use and there are no longer any references to the clustered object [10]. An algorithm called read-copy update is used by K42 to determine when all parts of the system have finished using a clustered object [18].

Read-copy update keeps track of which threads are able to access the object. When an object is submitted for destruction, a flag is set in the global translation table indicating that the object is waiting to be destroyed. Although the object has been marked for destruction, existing threads may still access representatives that have already been instantiated. The read-copy update algorithm tracks which threads can access the object, and determines that the clustered object is safe to destroy when all of the threads that can access the object have finished running. After it is determined that the object can be destroyed safely, a flag is set in the global translation table indicating that it is being destroyed. Then all entries for the object in the local translation tables are set to point to the default object and the clustered object's root is called to delete all of the representatives of the object and then itself. Finally, the global translation table entry for the object is marked as being unused.

The interaction of interposition and object destruction raises two challenges. First, when an interposed object is destroyed, it is necessary to remove it from its target. Second, destruction of target objects is a challenge, since the interposed object must correctly identify and forward all of the actions required for object destruction that it intercepts.

Chapter 4

Design and Implementation

This chapter describes the design and implementation of an object oriented interposing subsystem for K42. First, requirements for the interposition facility are presented. Then the design is described in three stages: (i) high level overview, (ii) discussion of implementation issues and (iii) low level implementation details. A number of Arbiters that have been implemented are presented in section 4.5.

The interposing subsystem allows objects, called Arbiters, to be interposed on function calls to a target clustered object in K42. Once an Arbiter has been installed, the Arbiter will obtain control whenever a caller calls a function of the target object. Once it has control, the Arbiter may decide to override the function of the target object by executing its own code and then returning control back to the caller, or it may decide to invoke the called function of the target object after executing its own prelude code and then executing postlude code after the call.

4.1 Requirements

The primary requirement of the interposition subsystem in K42 is the ability of an Arbiter to intercept all function calls to a target object and for the Arbiter to be able to call through to the originally targeted function. Other requirements include (i) transparency to the target object and the caller of the target object, (ii) efficient operation, (iii) integration with K42 clustered objects, (iv) ease of use for both programmers writing Arbiters and for users installing Arbiters, and (v) flexibility to use Arbiters throughout the K42 operating system and in applications running on K42. These requirements are elaborated further in this section.

Transparency: Transparency to both the target object and the caller of that object means that neither should require code modification to allow Arbiter interposition. Reducing the exposure of the interposition subsystem, both in changes to the program environment and in overhead such as execution time and

memory footprint of the Arbiter, is also desirable. Achieving complete transparency is impossible. For example, execution overheads or changes to memory locations may be detectable by the caller. However, these should be kept to a minimum

Efficiency: With respect to efficiency, the interposition subsystem must be efficient enough to be usable for a wide variety of applications, such as performance monitoring and hot swapping of objects. Inefficient interposition would result in perturbations that may be unacceptable for some applications. Moreover, poor performance would also hinder applications of interposition related to performance improvements. In K42, there are two aspects of efficiency that must be considered. The first is low overhead on a single processor. The second is low overhead when run on a multiprocessor system. Since one of the goals of K42 is excellent performance on large multiprocessor computers, both of these types of efficiency are important to the K42 interposition subsystem. Although large overheads on a single processor will translate to large overheads on a multiprocessor, maintaining good performance on a highly parallel computer can introduce additional problems due to contention on shared resources, even if interposition is efficient on a uniprocessor.

Integration with K42 Clustered Objects: Integration with the K42 clustered object system implies two things. First, the interposition subsystem must be able to interpose on K42 clustered objects. This is a requirement because clustered objects are used to encapsulate all resources and services in K42. Clustered objects are composed from multiple (at least two and possibly many) C++ objects, but to users they logically appear as one object. It is important that the interposing subsystem continues to present to users the familiar view of a clustered object being just one object. The second implication is that the interposing subsystem should itself be implemented using clustered objects, and in particular, the Arbiters themselves should be clustered objects. This is important because it maintains consistency in the system. Implementing the interposing subsystem in some other way would violate users' expectation of consistency, making it harder to use and maintain.

Ease of Use: Ease of use is another requirement for the K42 interposition subsystem. In particular, it must be relatively easy to write Arbiters. While Arbiters to perform sophisticated tasks may be complex, Arbiters that do simple things must be simple to write. Moreover, it must be easy to interpose an Arbiter on a target clustered object so that users are willing to interpose. It is envisioned that the interposition subsystem makes available a function that takes a target object reference as a parameter and hides all of the complexities of interposition.

Flexibility: The flexibility requirement is to ensure that as many tasks as possible can be implement using an interposition strategy. Two things are necessary for flexibility. First, it must be possible to interpose

on all significant components of K42 in the kernel and system servers as well as the K42 system libraries that reside in the address space of each application, and any clustered objects that are part of the application. Secondly, there should be as few restrictions on Arbiter code as possible. For example, interposed code must run with the same permissions and restrictions as the thread that called the object that is interposed on.

4.2 High Level Design

The K42 interposition subsystem design described here enables programmers to write Arbiter objects that users can then interpose on a target K42 clustered object. Once interposed, the Arbiter will intercept all function calls to the target object and execute its own code instead. At a high level, the interposition subsystem exploits the clustered object translation tables. To install an Arbiter, the appropriate entries in the (local and global) translation tables are overwritten so that they point to the Arbiter instead of the target object. All subsequent calls then automatically go to the Arbiter instead of the target object.

To simplify the writing of Arbiters, interposition functionality is split between the Arbiter object and a helper object called the ArbiterProxy. The ArbiterProxy is a generic object provided by the interposition subsystem and is used for all Arbiters (although there is flexibility to modify where required). The ArbiterProxy deals with the low level system complexities and is responsible for ensuring transparency. In particular, on interception, it saves all required registers (and other information), allocates a stack for the interposed Arbiter, and then calls the Arbiter.

The Arbiter is an object that has whatever domain specific functionality is required for the task it is supposed to perform and is written by programmers. The structure of an Arbiter is shown in figure 4.1. The handleCall() function, declared in lines 6-11 is called by the ArbiterProxy when it intercepts a function call. HandleCall() takes two parameters, a CallDescriptor object (described later in this chapter) that contains the information saved by the ArbiterProxy, and an unsigned integer that specifies which function in the target object's virtual function table was called. To transparently call the target object should it wish to do so, the Arbiter calls the makeCall() function, as in line 8 of the figure, which takes the same parameters as handleCall(). After makeCall() returns, the Arbiter optionally executes postlude code and then returns. Control returns to the ArbiterProxy object, which after restoring the environment, returns control back to the original caller.

Completing the description of the code in figure 4.1, line 3 declares the Arbiter's clustered object root

```
1  class RepArbiterPassthru : public CObjRepArbiter{
2  protected:
3          friend class CObjRootArbiterTemplated<RepArbiterPassthru, CObjRepArbiterProxy>;
4          RepArbiterPassthru() {}
5          DEFINE_LOCALSTRICT_NEW(RepArbiterPassthru);
6          virtual SysStatus handleCall(CallDescriptor* cd, uval fnum){
7                  // Arbiter specific prelude code
8                  SysStatus rc = makeCall(cd, fnum);
9                  // Arbiter specific postlude code
10                 return rc;
11         }
12 public:
13         static RepArbiterPassthru** Create(){
14                 return CObjRootArbiterTemplated<RepArbiterPassthru,
15                                                 CObjRepArbiterProxy>::Create();
16         }
17 };
```

Figure 4.1: Example Code for ArbiterPassthru

```
1  ArbiterRef arbiter = ArbiterType::Create();
2  DREF(arbiter)->captureTarget(co);
```

Figure 4.2: Interposing on a Target Clustered Object

class[1] as a friend of the Arbiter; this must be done in the Arbiter rather than in the base Arbiter class, since the root class is customized. The remaining lines are present to conform to clustered object conventions. Line 4 defines an empty constructor and line 5 defines a protected new operator. These lines are necessary to allocate memory for the representative from the appropriate memory pool and to prevent clustered object representatives from being created outside of the clustered object framework. Lines 13-15 define a static Create() function that is used to create a new Arbiter. It calls a corresponding static Create() function in the Arbiter's clustered object root that creates a new Arbiter root and registers it with the clustered object system.

[1] Clustered object roots were described in Chapter 3.

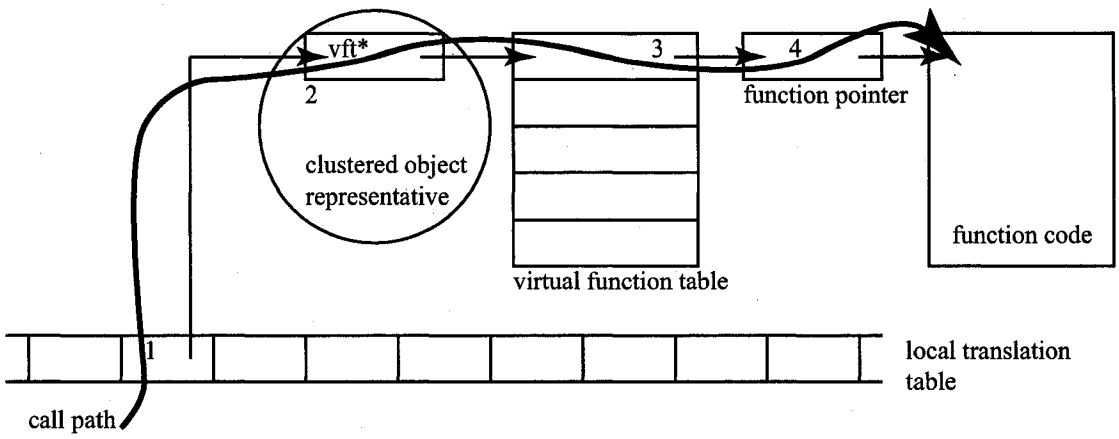


Figure 4.3: Layers of Indirection in K42

Arbiters provide two additional functions: captureTarget(), which interposes the Arbiter on the target object, and releaseTarget(), which stops the Arbiter from interposing on any more function calls. CaptureTarget() takes a single parameter, the clustered object reference of the target object to interpose on. ReleaseTarget() does not require any parameters. Both of these functions, CaptureTarget() and ReleaseTarget(), are inherited from the CObjRepArbiter class and do not need to be redefined or redeclared in the specific Arbiter.

4.2.1 Interposing on a Clustered Object

As described in chapter 3, the clustered object system uses indirection tables on each function call to locate the target clustered object and the appropriate representative. By replacing the target object's entries in the translation tables with entries for the ArbiterProxy, the ArbiterProxy is invoked instead of the target object. The ArbiterProxy object interposes the Arbiter and run its code in place of the target object's function. 4.2 depicts code that is typically used to interpose an Arbiter on a target clustered object. The first line creates an Arbiter of type ArbiterType using the ArbiterType's Create() function and initializes it as having no target. The Create() function is specific to each type of Arbiter, and an Arbiter author may add further initialization. The second line instructs the Arbiter to capture the target clustered object, co.

All calls to clustered objects are made by using the DREF() macro. A call then goes through four levels of indirection, shown in figure 4.3. Clustered objects are identified by a pointer that indexes into the local translation table, where a pointer to the appropriate local representative is found. Hence, the first level of indirection is through the local translation table in the clustered object system, which is used to locate the

correct representative for the clustered object. The second level of indirection is generated by the C++ compiler in the clustered object representative and is used to locate the virtual function table. The third level is through the virtual function table, and the fourth level of indirection is a pointer to the function code.

The interposition subsystem uses the first level of indirection in the local translation table to interpose on target objects. All calls to the public interface of a clustered object go through the local translation table in order to locate the correct clustered object representative. By overwriting the target object's local translation table entries with pointers to ArbiterProxy representatives, function calls to the target object can be redirected to the ArbiterProxy, which in turn calls the Arbiter.

To install the ArbiterProxy representatives in the local translation tables, the ArbiterProxy's miss handler is entered in the global translation table in place of the target object's entry and all local translation table entries for the target object are reset to point to the default object. This is shown in figure 4.4. On the left side of the figure, the Arbiter has been instantiated but not interposed. A target clustered object is also shown. On the right side of the figure, the Arbiter has interposed on the target object. The Arbiter calls a function in the clustered object system to substitute the ArbiterProxy miss handler for the target object's miss handler in the global translation table and reset the local translation table entries. The local translation table entries are reset to point to the default object, as is standard in K42, by sending a message to each processor to tell that processor. The target object's clustered object reference is passed with the message so that the correct local translation table entry is reset. After performing the substitution and resetting the local translation table entries, the clustered object system returns a pointer to the target object's miss handler, which is cached by the Arbiter for later use.

Then, as discussed in the previous chapter, when the default object is accessed, the clustered object system will call the miss handler installed in the global translation table to obtain and install a local representative before forwarding the call to the representative. Thus, after the ArbiterProxy miss handler has been installed in the global translation table and the local translation table entries have been reset, a call to the target object will ultimately result in an ArbiterProxy representative being installed and subsequently called.

Note that with this scheme is it possible to interpose multiple Arbiters on a target object, with the Arbiters being effectively stacked. The last Arbiter installed will have its ArbiterProxy's miss handler installed in the global translation table, and its target will be the ArbiterProxy of the Arbiter installed immediately prior to it. Each subsequent Arbiter will likewise have as its target the ArbiterProxy of the Arbiter installed immediately prior to it, except for the first Arbiter installed, which will have the original target clustered

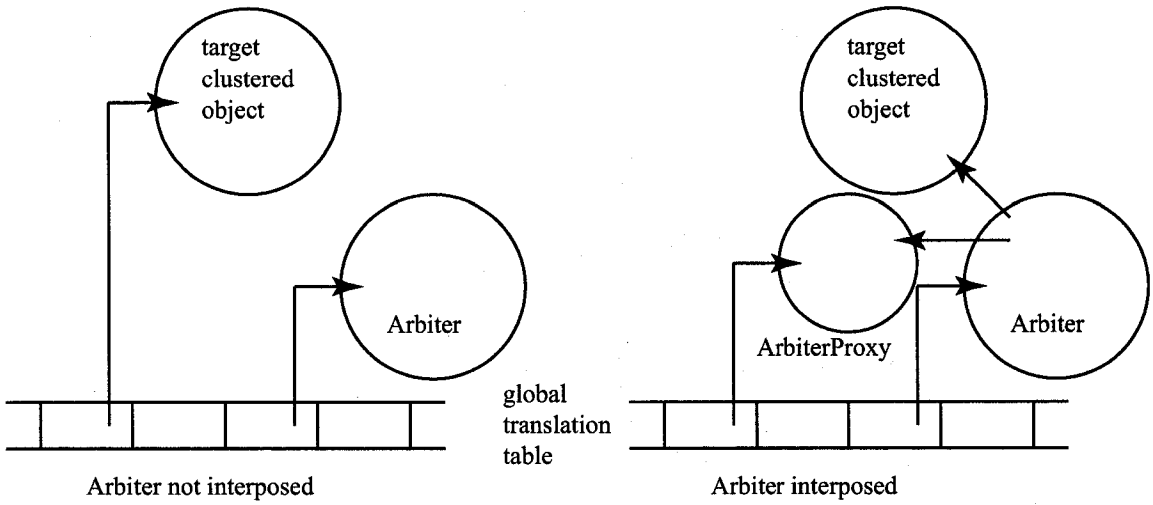


Figure 4.4: Interposing Using Translation Tables

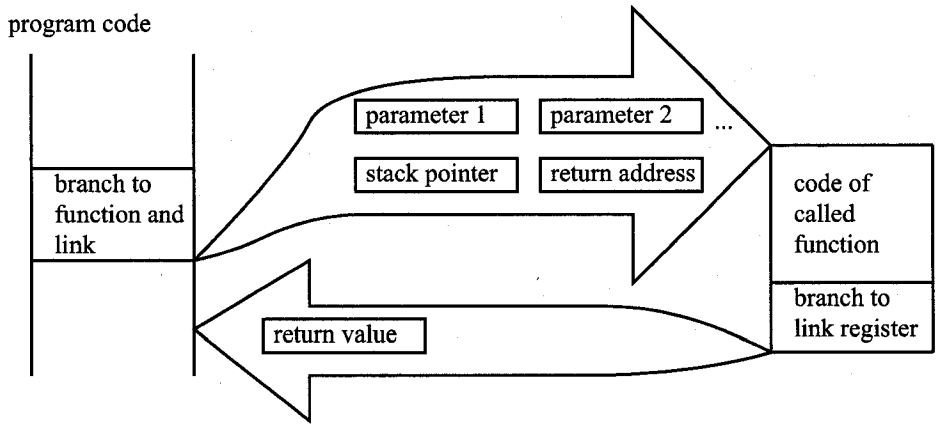


Figure 4.5: Data Involved in a Function Call

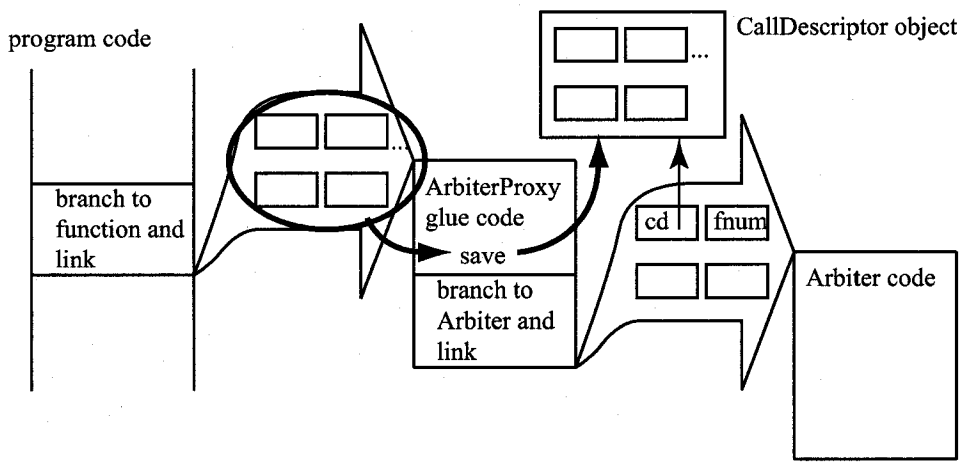


Figure 4.6: Arbiters Package Function Call Data

```
1  DREF(arbiter)->releaseTarget();
```

Figure 4.7: Removing an Arbiter from its Target

object as its target. Through the mechanism described above, an intercepted call will initially be directed through the translation tables and go to the last Arbiter installed. When each Arbiter in turn calls handleCall(), the call will pass through the chain of Arbiters, from last installed to first installed, and then to the original target clustered object.

4.2.2 Role of the ArbiterProxy

The ArbiterProxy is a clustered object. Its role is to stand in for the target object, assuming the role of miss handler, and when a call is made it (i) transparently saves state related to the call and existing execution environment, (ii) sets up an execution environment for the Arbiter, and then (iii) transfers control to the Arbiter. After the Arbiter returns, the ArbiterProxy tears down the execution environment and returns control to the caller. Each ArbiterProxy instance serves a single Arbiter.

When an ArbiterProxy is created, it is initialized with several temporary stacks and the reference of the Arbiter instance that it is to call. The ArbiterProxy has only a single function, namely arbiterMethodCommon(). Any call to a target object that has an Arbiter interposed is intercepted by a stub that determines which function was called and jumps to the ArbiterProxy's arbiterMethodCommon() function. This process is described in detail in the next subsection.

4.2.3 Intercepting Function Calls

The call to an interposed object is sent to an ArbiterProxy representative via the translation tables. The ArbiterProxy has a set of generic stubs that are installed into its virtual function table. They record the index in the virtual function table of the function that was called and then call a function in the ArbiterProxy to save information about the function call.

Figure 4.5 expands on the details of a function call. A call to a function is made with a branch and link instruction, which jumps to the function and saves the address of the next caller instruction in a register. *Call information,* such as parameters, a stack pointer, and the address for the function to return to upon completion, are located in well known places, such as in registers and on the stack . The compiler lays out this information so that each function knows what information is available and where to find it.

When a call is made to a clustered object with an Arbiter installed, the ArbiterProxy first acquires a

temporary stack. The temporary stack is necessary for transparency, because the Arbiter can not run on the same stack as the target function without facing unacceptable restrictions (described later). The ArbiterProxy then packages the call information into a CallDescriptor object, which it places on the temporary stack, before calling the Arbiter object that handles the call. The CallDescriptor object is an object that contains space for saved parameters, a stack pointer, return address, and any other information that needs to be saved on the architecture it is running on. This is shown in figure 4.6. Packaging the call information into the CallDescriptor must be done carefully, as registers can not be overwritten and used until after their contents are saved. After packaging the relevant call information into the CallDescriptor object, the ArbiterProxy calls the Arbiter's handleCall() function to execute the base functionality of the Arbiter. When handleCall() is called, it is passed the CallDescriptor object, allowing it to examine or change that information, along with a number identifying which function of the target object was called. HandleCall() passes control back to the ArbiterProxy object when it returns. At this point the ArbiterProxy releases the alternate stack that it had acquired and then returns to the caller.

4.2.4 Calling the Target Object

To call through to the target object's function, should it decide to do so, the Arbiter must locate a local representative of the target object, create an environment that the target function can run in, including restoring the saved call parameters on the stack and in appropriate registers, and then call the target function. To do this, the author of an Arbiter calls the makeCall() function, defined in the CObjRepArbiter class. MakeCall() restores the original execution environment and performs the call to the target object. The most important change to the execution environment to ensure transparency is to restore the original stack before the target function is called, and then reverting back after the call returns. Care has to be taken to do this in a way that is transparent to the target function. The saved parameters for the function call are obtained from the CallDescriptor object. After the target function returns, makeCall() restores the Arbiter's execution environment before continuing.

It should be noted that local representatives from target objects can not be obtained by calling through the local translation tables, since the relevant entries point to ArbiterProxy representatives. Similarly, the global translation table entry points to an ArbiterProxy miss handler. To locate a local representative, the Arbiter must have saved the target object's miss handler when the Arbiter was installed. This miss handler is called when the Arbiter needs to locate a local representative for the target object. Once obtained, the Arbiter representative caches it for faster access if it is called again.

4.2.5 Removing an Arbiter

Arbiters are removed by calling the releaseTarget() function of the interposed Arbiter, shown in figure 4.7. In most cases, removing an Arbiter is similar to interposing an Arbiter; however, removal can be more complex if there are multiple Arbiters installed on a single target object.

In the simplest case, when no other Arbiter is interposed on the target or when the Arbiter being removed is the last Arbiter that was interposed on the target, removal is done by switching the ArbiterProxy miss handler in the target's global translation table entry back to the target object's miss handler that the Arbiter stored when it interposed. To complete the removal, the local translation table entries for the target object are reset to point to the default object. A subsequent call to the target object will result in a translation miss, causing the local representative to be installed in the local translation table before the call is executed.

When a second Arbiter is interposed in front of the Arbiter being removed, the removal of the first Arbiter is done by replacing the target miss handler stored in the second Arbiter with the target object miss handler from the first Arbiter, and resetting the local representatives cached by the second Arbiter. Initially the second Arbiter's target will be the first Arbiter, and by changing the miss handler and resetting the local representative caches, the first Arbiter removes itself from the stack of Arbiters installed, and instead makes the second Arbiter have the target that the first Arbiter originally had. Arbiters can tell if another Arbiter has interposed on them by examining the miss handler stored in the target object's global translation table entry. If the installed miss handler belongs to their ArbiterProxy object, then there is no Arbiter interposed on them. If the miss handler does not belong to their ArbiterProxy object, then the Arbiter knows that a second Arbiter has interposed after it has. This detection and removal process is described in detail in subsection 4.3.5.

4.3 Design Decisions

This section discusses some of the more important design choices that were made in the design of the K42 interposition subsystem.

4.3.1 Interposing Using Translation Tables

The local translation tables are a good place to interpose Arbiters because they allow calls to a single target clustered object instance to be redirected to Arbiter representatives directly, without any modification to the representatives of the target clustered object or to the calling function and without affecting any other

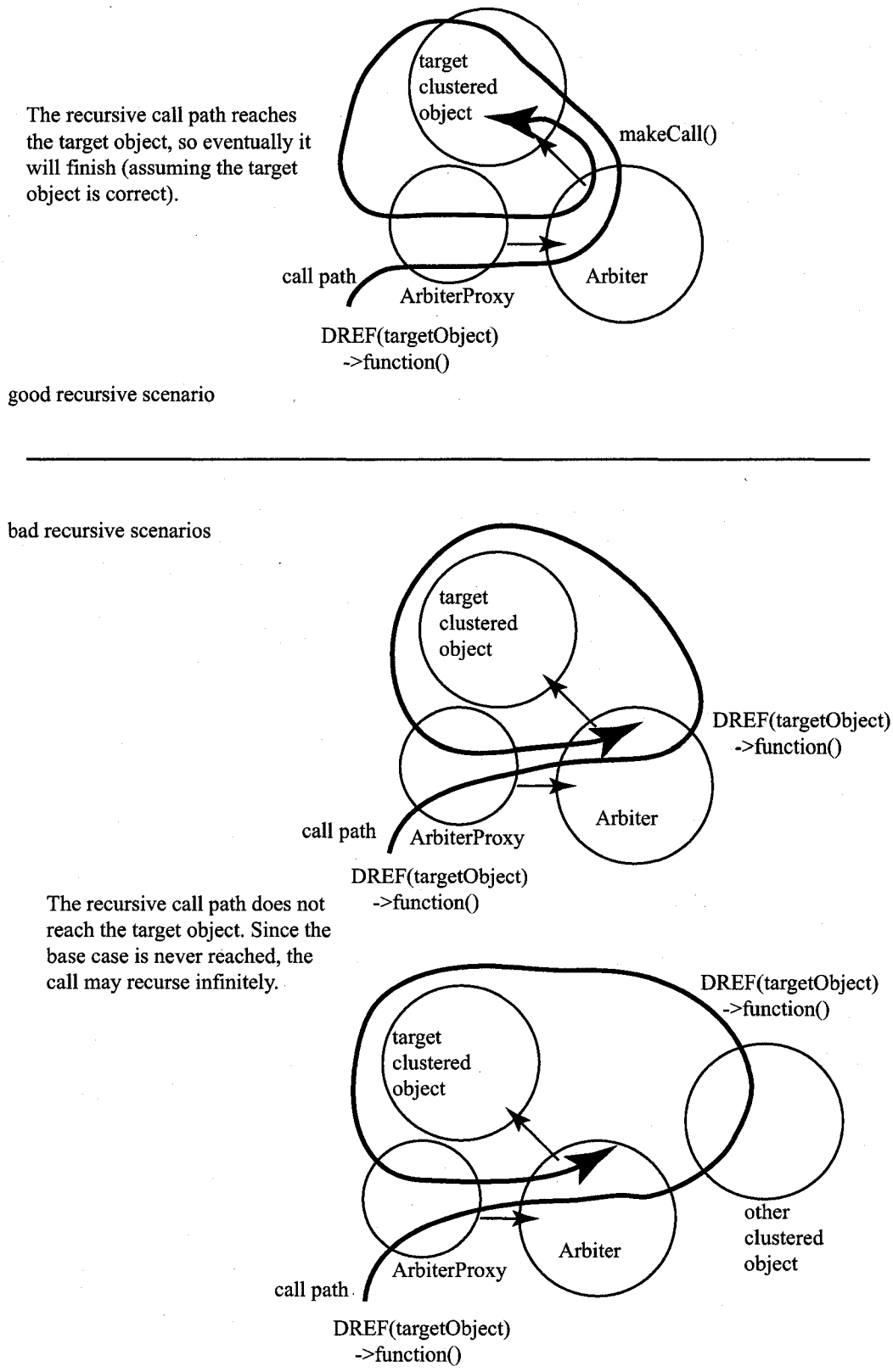


Figure 4.8: Recursive Scenarios Involving Arbiters

clustered object instances. This permits interposing to be done on a per-object instance basis. In contrast, indirection through the virtual function table or function pointers, the third and fourth indirection points in figure 4.3, would interpose on every object that used the virtual function table, which would be every instance of the object class. Indirection through the virtual function table pointer stored in the object, the second indirection point in figure 4.3, would require modifying every representative of the target object.

When the Arbiter is installed, the target object's local translation table entries are reset to refer to the default object described in chapter 3, and its miss handler in the global translation table is overwritten with a replacement miss handler belonging to the ArbiterProxy. Since the local translation table entries are reset, when a call is made to the target object, the ArbiterProxy's replacement miss handler is called to install an ArbiterProxy representative. While this adds the overhead of handling a miss on every processor the target object is called on after the Arbiter is interposed, this overhead is only caused on the first such call.

Interposing on the translation tables introduces a potential for infinite recursion that users of Arbiters must be aware of and avoid. If a clustered object is recursive, and an Arbiter interposed on it calls the target object with makeCall(), then the Arbiter will be called with each recursive call without any problems. The top part of figure 4.8 shows this case. At some point the target object will reach its base case and return, and the call stack will unwind.

On the other hand, if the Arbiter makes a call to the target object in the standard K42 way using the DREF macro, then unintended infinite recursion on the Arbiter will result, as depicted in the center part of figure 4.8. The target object is never reached because the Arbiter continuously intercepts the call to the target object that used DREF, leading to infinite recursion. In fact, the Arbiter is even further constrained in that it may not call any function of any object that might invoke a function of the target object. This scenario is shown in the bottom of figure 4.8, where the target object is never reached, because calling the other clustered object indirectly leads to infinite recursion.

4.3.2 Role of the ArbiterProxy

The ArbiterProxy has two roles, first as a miss handler to stand in for the target object, and second, to process intercepted calls and then pass them to the Arbiter to handle. As a miss handler, there are distinct advantages to having the ArbiterProxy be a proper clustered object with its own root and representatives. In its role in intercepting function calls, the ArbiterProxy must transparently intercept and package function call details, set up an execution environment for the Arbiter, and then call the Arbiter.

The primary purpose of introducing the concept of a separate ArbiterProxy object is to simplify the

structure of the Arbiter object for programmers. The ArbiterProxy object encapsulates all of the common and typically low level code required to intercept function calls, save their state and set up an execution environment for Arbiters to run in. This includes saving floating point and integer parameter registers, and the link register that specifies where execution will resume after the Arbiter returns.

The advantage of implementing the ArbiterProxy as a clustered object is that this approach does not require miss handlers to take different actions based on which translation table they are handling a miss for. This avoids problems with target object miss handlers, since target objects install their local representatives in dummy local translation table entries contained in the Arbiter. The dummy entry looks exactly like the real entry, so this is transparent to the target object. If there was no ArbiterProxy, the miss handler would not be able to correctly detect which translation table entry was causing the miss. If clustered object miss handlers could take different actions for different translation table entries, then problems could arise if a dummy local translation table was used by an Arbiter trying to locate a local representative of the target object.

In summary, when intercepting calls, the ArbiterProxy transparently intercepts calls made to the target object and redirects them to the Arbiter. When it intercepts a function call, the ArbiterProxy's arbiterMethodCommon() function is called automatically, which saves the function parameters and other call state. The ArbiterProxy also acquires a temporary stack and initializes it, and then calls the Arbiter's handleCall() function to take whatever action is appropriate to deal with the function call.

4.3.3 Multiple ArbiterProxy Objects with a Single Target

When multiple Arbiters are interposed on the same target object, each Arbiter installs its own ArbiterProxy object. An alternative strategy would be to instead use a single ArbiterProxy were used for all Arbiters. In that case, the Arbiter would have to know when interposing if there was already an Arbiter installed. If there was no Arbiter installed, it would create and interpose an ArbiterProxy object as described previously. If there was already an ArbiterProxy object, the Arbiter would register with the ArbiterProxy object and get placed in some position in the chain of interposed Arbiters. When called, the ArbiterProxy would save the call state, and then call the first Arbiter in the chain. After that, each Arbiter's makeCall() function would call the next Arbiter in the chain, if there was one, and if there were no Arbiters remaining in the chain, the Arbiter would restore the target object's execution environment and call it. There are two advantages to using only one ArbiterProxy object per target object. First, there would be a reduction in the amount of memory used when multiple Arbiters are installed. Each ArbiterProxy keeps a pool of stacks, and

having fewer stack pools would reduce memory usage. Second, there would be a reduction in the overhead of multiple Arbiters, since the time required to restore and save the target object's execution environment each time an Arbiter is called would be eliminated. A disadvantage of using a single ArbiterProxy object is that the ArbiterProxy class could not be specialized, since it would have to be suitable for all possible Arbiter types. Using just a single ArbiterProxy for multiple Arbiters would add too much complexity in relation to the advantages.

Performance results (presented in Chapter 5) show that the primary overhead of changing execution environments is obtaining an alternate stack when needed. This overhead only occurs once per call, since multiple Arbiters can reuse the same stack, so the efficiency savings are minimal. Due to the flexibility gained by being able to specialize the ArbiterProxy if needed, and considering that the case of multiple Arbiters interposed on the same target object is expected to be uncommon, it was decided that each Arbiter should install its own ArbiterProxy instance, even though the memory requirement is larger for more stack pools when there are multiple ArbiterProxy objects.

4.3.4 Interactions between Multiple Arbiters with a Single Target

When multiple Arbiters are interposed on the same target object, interactions between Arbiters may occur in three cases: (i) interposition of an additional Arbiter, (ii) making a call through to the next Arbiter and (iii) removal of an Arbiter. The first two cases of interaction are reasonably straight forward given the design, and they are discussed in this section. The third case, removal of an Arbiter is more complex when multiple Arbiters are involved, and is discussed in the next section.

Interposing multiple Arbiters on a single clustered object is straightforward, with the current design. Arbiters are able to interpose on any clustered object, and interposing on a clustered object that is already interposed on will simply cause the second Arbiter to interpose on an ArbiterProxy object instead of the original target object. As a result, Arbiters are effectively stacked, with the first Arbiter interposed referring to the target object, and the last interposed Arbiter being referred to by the translation table. The Arbiter installation process is protected by a lock, so that multiple Arbiters attempting to interpose on the same clustered object at the same time will not result in any race conditions. The procedure used is exactly the same for the case involving multiple Arbiters as it is when there is only one Arbiter.

One potential problem is that is that the same type of Arbiter can be interposed on the same target multiple times (perhaps by different clients). In some cases, such as performance monitoring, it may be preferable to have only one Arbiter to perform the monitoring and report it to multiple clients, rather than

have several Arbiters gather the same monitoring information. Currently this issue is not addressed since it is a performance issue rather than a correctness issue. Arbiters will work properly with multiple identical Arbiters installed.

When a call is made to an object that has multiple Arbiters interposed, there are several issues to consider. The first issue is correctness. If multiple Arbiters are interposed then there will be a chain of ArbiterProxy and Arbiter objects leading to the target object. Each ArbiterProxy installed after the first interposes on the previously interposed ArbiterProxy object. With the current design, this happens in a way that is transparent to the previously interposed ArbiterProxy objects. Moreover, when an Arbiter calls its target, it does not know whether the target is the actual target or another Arbiter. Overall, this works as expected, presenting no problem to the interposing Arbiter or to the previously interposed Arbiter.

The second issue is one of semantics. If there is a chain of Arbiters leading to the target, then any of the Arbiters may decide not to call through (and return instead). Hence, it is possible that an Arbiter in the chain may not be called even though a client has invoked the target. It is also possible that an Arbiter (early in the chain) is invoked even though in the end the target object is never called. As a result, interposers may want to have precise control of the order of the Arbiter chain. However, this is not currently supported.

4.3.5 Removing Arbiters

Removing Arbiters is significantly complicated due to the possibility of there being multiple Arbiters installed. and a number of scenarios can lead to complications. For example, if the Arbiter being removed restored the clustered object that was present when it was interposed, then the remaining Arbiters might no longer be called. As another example, if one were to simply and naively remove an Arbiter from the chain, then one of the remaining Arbiters may be left with a reference to an Arbiter that had been removed. If an Arbiter is removed from the chain, and possibly destroyed, and then called, the results would be unpredictable, and could lead to a system crash.

Some of these scenarios are shown in figure 4.9. The left side of the figure shows three Arbiters interposed on a single target object. The simplified diagram represents each Arbiter and ArbiterProxy pair with a single circle. When an Arbiter is removed, it naively restores the miss handler of its target into the global translation table. In the middle part of the figure, Arbiter A is removed, and sets the target object's translation table entries to reference A's target, which is the original target object. Note that Arbiters B and C are not included in the new chain of interposed Arbiters, but still believe themselves to be interposed. In the right hand side of the figure, Arbiter C is removed. It also restored the miss handler of its target into the

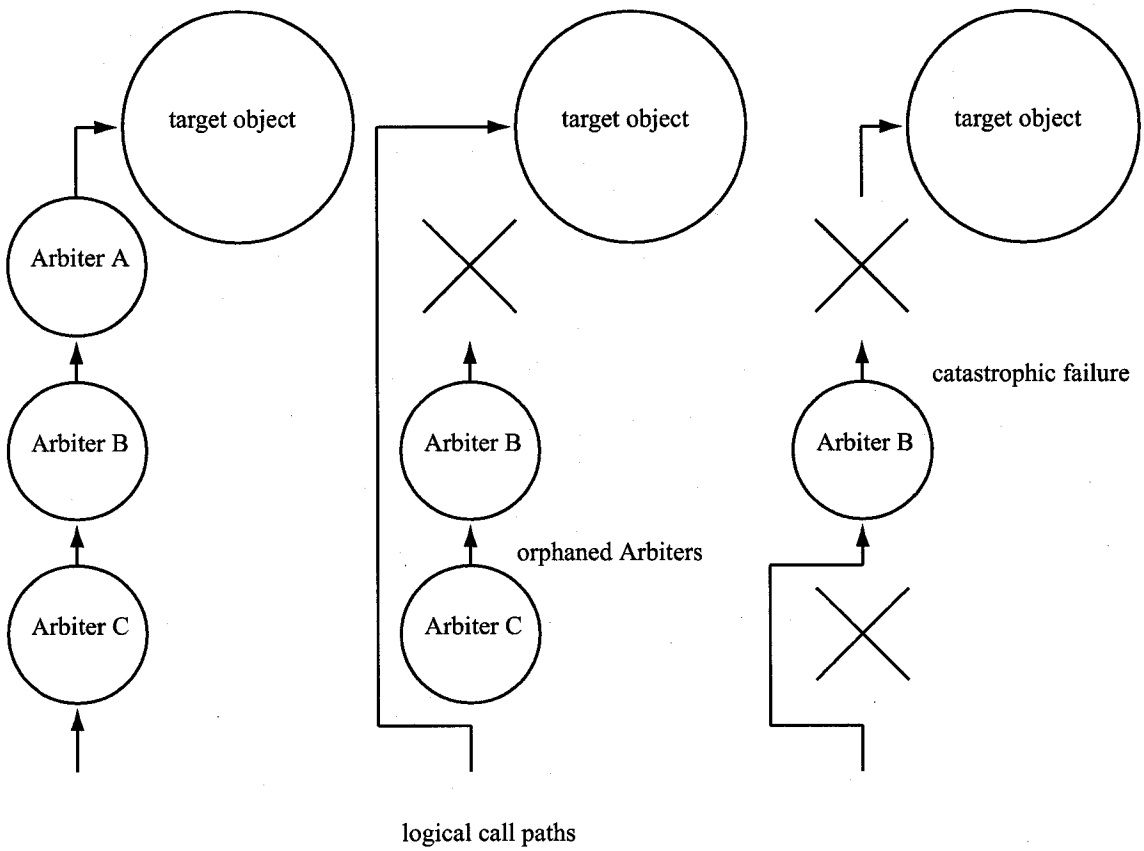


Figure 4.9: Failure due to Improper Arbiter Removal of Multiple Arbiters

translation tables, and the target object's translation table entries refer to Arbiter B. Catastrophic failure can result when Arbiter B intercepts a call and attempts to call its target, the now-destroyed Arbiter A.

To solve these problems, multiple Arbiters interposed on the same target object are treated as a list, with the translation tables pointing to the first interposed Arbiter, and each Arbiter containing a pointer to either the target object or the next Arbiter in the list. The algorithm is shown in figure 4.10. Conceptually, removing an Arbiter is equivalent to removing an object from the list of Arbiters. This requires obtaining the pointer to the head of the list from the translation table, with a special case if the Arbiter being removed is the head. Otherwise, the list of Arbiters is followed, maintaining a reference to the previous Arbiter until the Arbiter being removed is found. The pointer to the previous Arbiter is then swung to the target of the Arbiter being removed, removing the Arbiter from the list. Complications due to concurrency and other details are discussed in the next section.

Alternatively, a doubly linked list could be kept by keeping a backpointer to the previous Arbiter inside of each Arbiter. This would eliminate the need to scan through the list of Arbiters. It would, however,

```
1  // get head of Arbiter list
2  currArbiter = translation table entry
3  if currArbiter != ArbiterToRemove
4          // find the Arbiter to remove
5          while currArbiter != ArbiterToRemove
6                  prevArbiter = currArbiter
7                  currArbiter = currArbiter->next
8          // remove Arbiter from list of Arbiters
9          prevArbiter->next = currArbiter->next
10 else
11         // special case for removing first (or only) Arbiter
12         translation table entry = currArbiter->next
```

Figure 4.10: High Level View of Removing an Arbiter

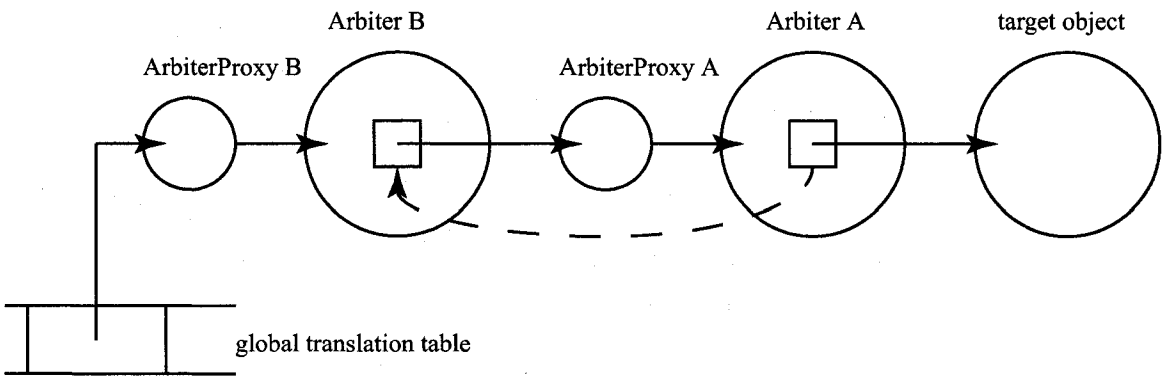


Figure 4.11: Removal Procedure for Multiple Arbiters Interposed on a Single Target

increase the complexity of interposing Arbiters, as it would be necessary to determine if there was another Arbiter installed at installation time, and to patch the other Arbiter's backpointer if that were the case. A doubly linked list of Arbiters would also require an additional special case to handle the tail Arbiter, as the target object would not have a backpointer.

Locating the head Arbiter when an Arbiter is removed is easily done by examining the miss handler of the target object. The target object's miss handler will be an ArbiterProxy object, which will contain a reference to its corresponding Arbiter.

Figure 4.11 shows the removal process in more detail, with Arbiter A as the Arbiter being removed and Arbiter B having been interposed after Arbiter A. Arbiter A knows how to find the location of Arbiter B from its ArbiterProxy object. This is done by calling the master() function, which is found in every

ArbiterProxy miss handler. Since each Arbiter stores the location of the target miss handler that it interposed on, Arbiter A can follow the chain of ArbiterProxy objects and Arbiters until it finds the one that interposed immediately after it did. (This is no different than the functionality required to remove an element from a single linked list.) In the example, Arbiter A sees that Arbiter B's target is its own ArbiterProxy. Arbiter A then replaces Arbiter B's target miss handler pointer with its own, removing itself from the chain. To complete the replacement, Arbiter A overwrites any pointers to target local representatives cached in the local representatives of Arbiter B.

The singly linked is used to chain Arbiters instead of a doubly linked list due to the simplicity. A doubly linked list would allow Arbiters to be more easily inserted anywhere in the list, which could help in solving the semantic problems discussed earlier. However, it would also be necessary to detect other Arbiters at install time to solve these issues. The singly linked list of Arbiters does solves the stability problem in Arbiter removal in simple manner, with a reduced number of special cases and no need to modify the Arbiter installation procedure. Unlike solutions using a doubly linked list, a singly linked list does not imply any solution to the additional problem of the semantics of multiple Arbiters, which were not comprehensively examined due to time constraints.

If there is a hot-swap in progress, then an object other than an Arbiter may have interposed. To avoid problems, Arbiters check for hot-swapping when they are installed or removed, and these operations fail if there is a hot-swap in progress, since the Arbiter can not deal with the hot-swap mediator. This issue is expected to be resolved shortly, as there is currently work underway to rewrite the hot-swap mediator as an Arbiter.

4.3.6 Object Destruction Interactions

The interactions of Arbiters during object destruction require careful design to ensure correct operation. Two scenarios are handled in order to avoid problems. The first scenario is when an Arbiter that is interposed is destroyed, and the second scenario is when a target clustered object that has some Arbiter interposed on it is destroyed.

K42 clustered objects can not be destroyed by a user; instead users must remove all permanent references to the object in the system and then submit the object to a garbage collection system. The garbage collection system waits until all threads that may be in flight and using the object to be destroyed have completed. After this happens, the garbage collection system safely destroys the object. To solve the first scenario, an Arbiter that is submitted to the garbage collection system immediately removes itself from its target object

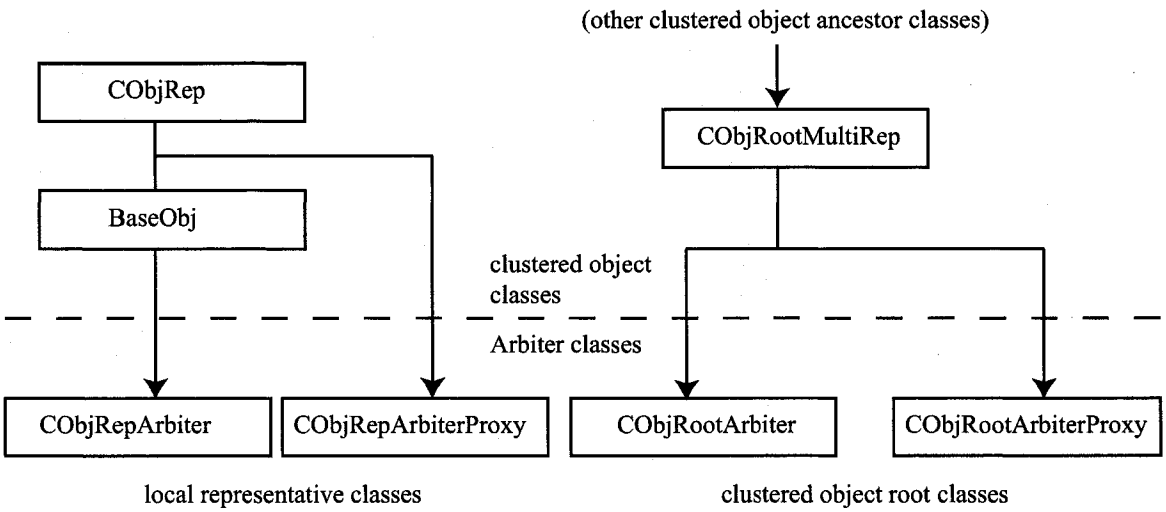


Figure 4.12: Arbiter Class Hierarchy

before submitting itself to the garbage collection system. This ensures that destroyed Arbiters do not remain interposed.

The second scenario, where a clustered object that is targeted by an Arbiter is being destroyed, is more complex. An Arbiter can block the destruction request, since it is intercepted by the Arbiter. Hence, it is important that the Arbiter not block the call to destroy the target clustered object it is interposing on. Requests to delete clustered objects go to the miss handler, which has a cleanup function. Clustered objects are deleted by calling the cleanup function of their miss handler. This call will be intercepted by an interposed Arbiter. When an interposed Arbiter intercepts a call to the target object's cleanup function, it does not delete itself. Instead, the Arbiter removes itself from the target object then forwards the cleanup call to its former target. This ensures that Arbiters do not remain interposed on destroyed clustered objects.

4.4 Implementation Details

In this section, the implementation of the K42 interposition subsystem is discussed in greater detail.

4.4.1 Arbiters as Clustered Objects

Arbiters and ArbiterProxies are implemented as clustered objects. They are part of the clustered object class hierarchy. The class hierarchy is shown in Figure 4.12. Since clustered objects have two parts, root and representative, the Arbiter and ArbiterProxy both have root (CObjRootArbiter and CObjRootArbiterProxy, respectively) and representative (CObjRepArbiter and CObjRepArbiterProxy, respectively) parts. Both

root objects are descendants of CObjRootMultiRep, which provides support for objects that have a different local representative for each processor. CObjRep is the standard base class for representative objects, and BaseObj adds support for interprocess communication so that objects can be called from external address spaces. The ArbiterProxy does not need this support, since it is not called directly by a user. Any interprocess communication support that is necessary will be done by the target object when it is called using makeCall().

4.4.2 Intercepting and Packaging Untyped Function Calls

Call state that must be saved on calls through an Arbiter include the parameters of the target function, the stack pointer, and return address of the callee. Since the ArbiterProxy can not know in advance what parameters the target object's function may take, it must save all potential parameters. The parameters are potentially located in integer registers, floating point registers, and the thread's stack. Stack parameters are left on the stack, and all potential parameter registers are saved in a helper object called the CallDescriptor. To do this, the interposing subsystem uses a solution modeled after the K42 default object and generic function, which stand in for any clustered object that does not have a representative on the processor they are being called on [20]. The ArbiterProxy representatives stand in for the target object's representative. Each ArbiterProxy representative consists of a virtual function table with entries that point to stub functions that record their index number and jump to the ArbiterProxy's generic function, called arbiterMethodCommon(). The generic function is written in assembly code and first saves the call state, and then call a stub that calls the Arbiter's handleCall() function to perform additional work.

The first source of state that must be saved is the set of registers that may contain parameters. On the PowerPC systems that K42 runs on, these are integer registers r4 through r9, and floating point registers f1 through f13. Integer register r3 is also a parameter register; however it is not saved because it contains the "this" pointer for the object. Since the object being called is the ArbiterProxy the "this" pointer in r3 refers to the ArbiterProxy, not the target object. The correct "this" pointer for the target object can be determined later. The second source of state is the stack. Hence the stack pointer must also be saved to allow the parameters on the stack to be located. The third source of state is the call itself; to handle the call the Arbiter must record the link register that specifies where to return to, and it must record the index number in the virtual function table of the function that was called.

To permit the saved state to be examined later or used for calling the target object, it is saved in the CallDescriptor. Since the amount of state is dependent upon the architecture that K42 is being run

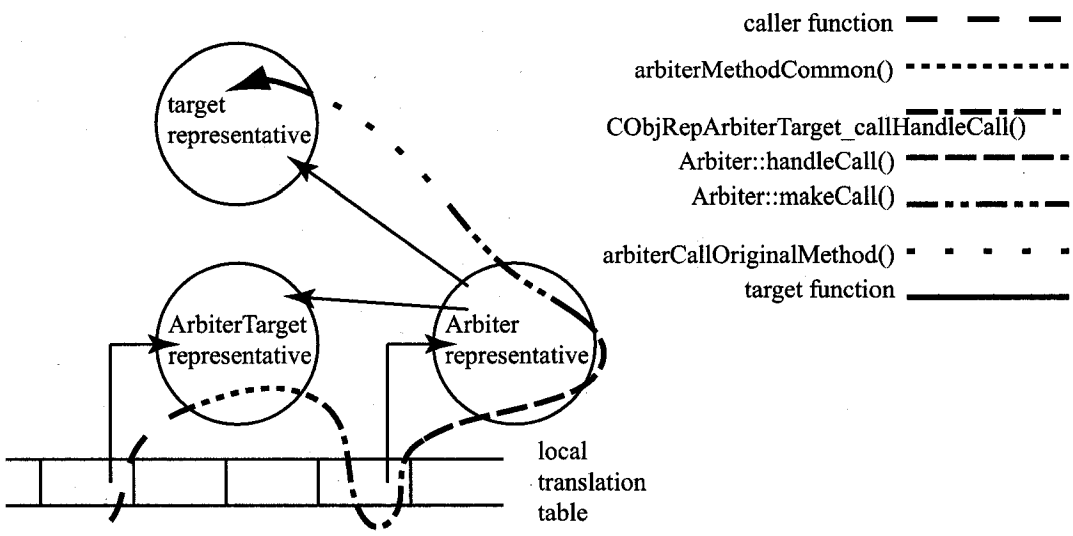


Figure 4.13: Calling the Arbiter's Target

on, the CallDescriptor has an architecture dependent portion that acts as a data store, and an architecture independent portion that serves as an interface to the state information. The architecture dependent portion stores all information that needs to be saved to memory, and has methods to read and write the parameters the function call was made with. The architecture independent portion allows the parameters to be read and written in a uniform manner on any platform, even though they may be in varying locations (i.e. in different locations on the stack, and in varying numbers of processor registers).

Upon return from the function in the Arbiter, the ArbiterProxy representative's generic function only needs to restore the original stack, return the alternate stack and return to the caller. The return value is always a K42 error code, and unless there is a failure in the ArbiterProxy, it just passes on the returned value. The registers that stored parameters for the target function do not need to be restored when returning, and the generic function does not use any other resources that need to be restored or returned.

4.4.3 Calling a Target Object

To properly call a function of the target object (from the Arbiter), it is necessary to first locate the representative of the target object on the current processor. Then it is necessary to restore the state associated with the original call, before making the call to the target object's function. When the call returns, it is necessary to restore the Arbiter's state. All of this is implemented in the makeCall() function so that the writer of the Arbiter need not concern himself with these details (in the common case).

Figure 4.13 shows the path taken by a call that is interposed on. The one function in the ArbiterProxy's interface is the generic function, arbiterMethodCommon(), an architecture-specific function that saves the function call state and sets up an execution environment for the Arbiter, as described previously. It then

calls CObjRepArbiterProxy_callHandleCall(), which is a helper function to call the Arbiter's handleCall() function in an architecture-independent way. This call is a normal clustered object function call, which means control flows through the local translation table. In the figure, the Arbiter's handleCall() function in the figure decides to call the target object as part of handling the call. It does so by calling the makeCall() function, which locates a local representative of the target object, using the target object's miss handler. MakeCall() calls a second function, arbiterCallOriginalMethod(), that takes care of the low level, architecture specific details of restoring the call parameters and setting up the target object's execution environment, before calling the target object. After the call has completed, control returns to arbiterCallOriginalMethod(), which restores the Arbiter's execution environment, then returns to the Arbiter, which continues with any other tasks that it would like to. After it has completed its work, the Arbiter returns, passing control back to the ArbiterProxy, that in turn passes control back to the caller.

For makeCall() to obtain a pointer to a local representative of the target object, each Arbiter representative can use the target object's miss handler that was stored when the Arbiter was interposed. In general, miss handlers would fill in the corresponding local translation table entry. In this case, the Arbiter passes the miss handler a dummy local translation table entry, which the miss handler fills and the Arbiter can use this value later to locate the local representative. The local representative pointer is thus cached so that it can be reused if it is needed again. Although the normal semantics of K42 are that pointers to local representatives should not be stored, the Arbiter can safely store these pointers. The Arbiter obtains the pointer itself, so it knows that it is a valid representative pointer, rather than a pointer to the default object. Since the Arbiter stores the pointer in its local representative on the same processor, it is in no danger of using it on other processors.

In the arbiterCallOriginalMethod() function, the Arbiter restores the call state that was saved in the CallDescriptor object when intercepting the call. This is similar to the steps taken when intercepting the original call, except in reverse. To restore the parameters, all parameters saved in the CallDescriptor object are restored to their original locations, and the original stack pointer is restored so that the target object's function will use the original stack. The alternate stack is saved in the thread descriptor so that it can be conveniently located when the target function returns without having to access thread-specific storage in the Arbiter in this very low level section of code, and as an optimization to allow it to be reused, as described in the following sections. In addition the "this" pointer for the target object needs to be determined. The "this" pointer for the original call comes from the representative that was obtained to handle the call, so the target's local representative is used as the new value. Then the target object's function is called.

After the call to the target object has completed and control returns to the Arbiter, arbiterCallOrigi

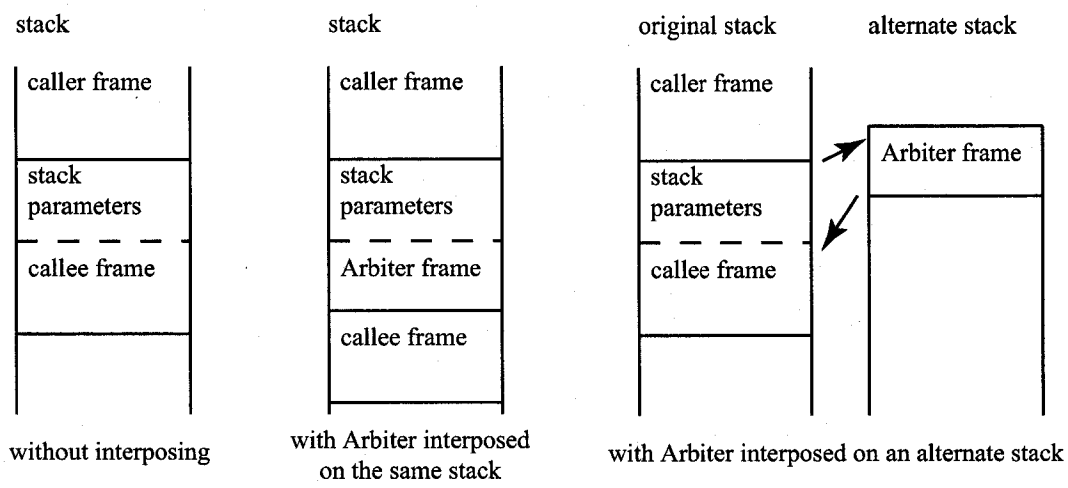
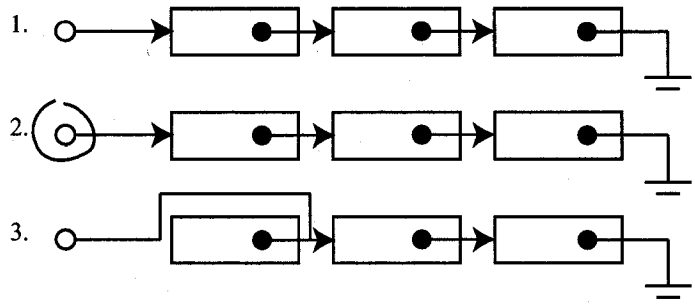


Figure 4.14: Parameters Passed on Stack

nalMethod() restores the call state required for the Arbiter to run. In particular, this means restoring the Arbiter's stack pointer. Copying parameters is not necessary on the return path because the only parameter is the return value. Since all clustered object functions have an error code as the return, this value is passed directly back to the Arbiter.

4.4.4 Locating an Alternate Stack

To permit the caller's stack and any parameters on it to be used by the target object, the Arbiter uses an alternate stack. The alternate stack is allocated and the switch is made by the ArbiterProxy before calling the Arbiter. Figure 4.14 shows the issue that is addressed by the alternate stack. The first part of the figure shows a stack that a caller and callee share without an Arbiter interposed. Parameters for the callee that do not fit in the parameter registers reside in the bottom part of the caller's stack frame. The next frame belongs to the caller, which is able to find the parameters directly above its own frame. The second part of the figure shows a naive implementation, in which an Arbiter is interposed on the callee using the caller's stack. The Arbiter's stack frame is in between the caller's frame and the callee's frame. This case presents a problem because the callee will get incorrect values when it attempts to access the stack parameters located directly above its own frame. Copying the callee parameters will not help, because the Arbiter can not know in any general way how many of them there are. The third part of the figure shows the chosen solution. Two stacks are used, one for the caller and the callee, and a separate one for the Arbiter. Instead of placing the Arbiter's frame below the caller's frame, the Arbiter's frame is placed on the alternate stack, and the callee's frame goes directly below the caller's frame. When the Arbiter calls the callee function, it must switch back to the original stack, and restore its own stack upon the callee's return.



4. retry from step 1 on failure

Store Conditional only returns success if no other thread performed a Load Linked or Store Conditional since the corresponding Load Linked instruction. Therefore the thread that succeeded can proceed, and other threads will fail and must retry.

Figure 4.15: Lock Free Linked List

```
1  if ThreadDescriptor->altStack == null
2          repeat
3                  stackHead = LoadLinked(stackList)
4                  stackNext = head->next
5          until stackList = StoreConditional(stackNext)
```

Figure 4.16: Algorithm for Acquiring an Alternate Stack

The thread descriptor has been augmented with a field to locate the alternate stack; the Arbiter uses this field to permit the same stack to be used for recursive calls to a clustered object or for multiple Arbiters interposing on calls to the same or different clustered objects that are called in the same thread. Without this field, each Arbiter would have to have some thread specific storage to locate stacks for recursion, and stacks would not be shared by multiple Arbiters.

When an Arbiter intercepts a call to its target object, the alternate stack is acquired by the arbiterMethodCommon() function. The Arbiter also restores the original stack when calling the target object's function and when returning to the caller. It switches back to the alternate stack when the target object's function returns. In this cases the alternate stack has already been acquired, so only switching is necessary and no new allocation is required.

Arbiters acquire alternate stacks from a fixed size pool of stacks that is maintained in each ArbiterProxy's local representative. To handle multiple processors efficiently, each ArbiterProxy has stacks allocated to each processor it has a local representative on. Replication of resources is a common approach in K42 and in this case allows acquisition of stacks without worrying about contention from other processors. Stacks are stored in the ArbiterProxy rather than the Arbiter because the number of available registers is

very limited, and storing stacks in the ArbiterProxy reduces the number of registers that must be spilled to memory to acquire a stack in the arbiterMethodCommon() function. Each ArbiterProxy representative has a pool of stacks because there may be several concurrent threads on the same processor. When a thread needs a stack for an Arbiter to run on, it gets one from the pool in the ArbiterProxy. The fixed size of the stack pool is a limitation of the current Arbiter implementation.

The ArbiterProxy's stack pool is implemented as a list of stacks, allocated when the ArbiterProxy representative is created. When an Arbiter obtains a stack, it removes it from the list of free stacks so that another Arbiter will not attempt to use it at the same time. To avoid locking when stacks are allocated or deallocated, the Arbiter uses lock free synchronization based on the PowerPC's load linked and store conditional instructions. Figures 4.15 and 4.16 illustrate how a stack is allocated. The Arbiter loads a pointer to a record containing information about a temporary stack from a memory address using load linked, creating a link for that address. It then reads the pointer to the next record and saves it back into memory using store conditional. If the memory at that address has been changed since the link was created then the link will have been destroyed and the store conditional will fail, causing the ArbiterProxy will retry.

4.4.5 Reusing Alternate Stacks

To prevent allocation of numerous stacks when an interposed target object is called recursively, threads only acquire one alternate stack for Arbiters and reuse it for recursive calls. Since the Arbiter acquires a new stack for each call, recursive calls to clustered objects that have an interposed Arbiter would otherwise acquire many stacks. Acquisition of multiple stacks by a single thread would not lead to synchronization problems or race conditions, but allocating a large number of stacks could waste significant amounts of memory. Using many stacks would also cause problems due to the limited number of stacks available in the current implementation. This problem is solved by having each thread descriptor contain a pointer to an alternate stack. Before attempting to acquire a new stack for an Arbiter to run on, the ArbiterProxy thread first checks the alternate stack pointer in the thread descriptor to see if it has already obtained a stack. If it finds one, it uses that stack.

Figure 4.17 shows the stacks of a set of recursive function calls with an interposed Arbiter that reuses the same alternate stack for each recursive call. Before calling the target object, the Arbiter pushes a frame on the alternate stack. It then restores the original stack, saves the pointer to the alternate stack in the thread's alternate stack record and calls the target function. Then, when the target clustered object calls itself, resulting in another call to the Arbiter, it uses the next frame on the alternate stack. This is repeated

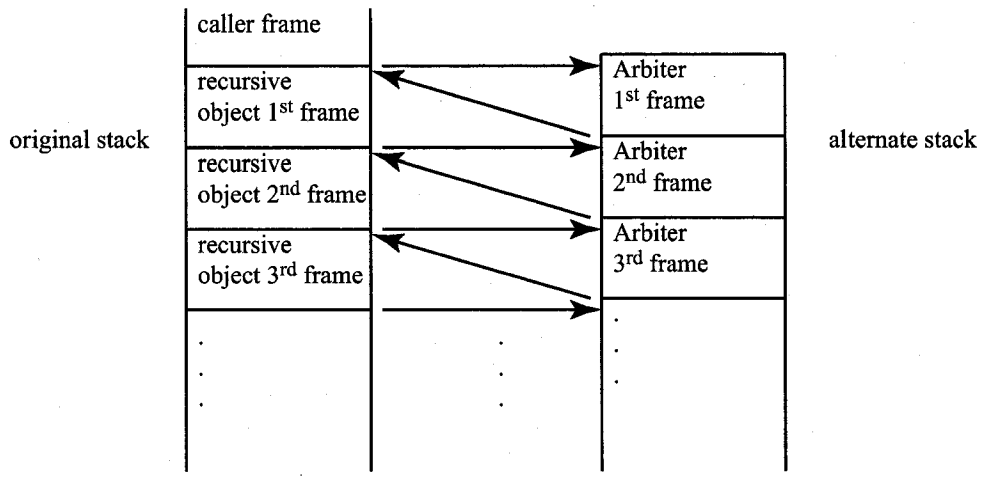


Figure 4.17: Alternate Stack with a Recursive Clustered Object

as many times as the recursive target clustered object calls itself. The Arbiter keeps track of the depth of recursion in a field that is stored along with each alternate stack so that the stack can be released when it is no longer in use.

The algorithm also reuses the stack when there are multiple Arbiters in use simultaneously; this can be when multiple Arbiters interposed on the same target object, or multiple Arbiters interposed on different target objects that are all invoked in a single call path. Different Arbiters can each have frames on the alternate stack if the target object makes a call that is intercepted by an Arbiter. This is shown in figure 4.18, in which Arbiter A calls its target, which through some chain of function calls invokes Arbiter B. Arbiter B finds Arbiter A's alternate stack already in place and uses it.

Figure 4.19 shows a further optimization that enables the original stack to also be reused when an Arbiter calls a clustered object that has an Arbiter interposed. For example, if Arbiter A is interposed on object A and Arbiter B is interposed on object B, and Arbiter A calls object B. In this case, Arbiter B will intercept the call and will use the original stack, rather than allocating a second alternate stack. To enable this, the original stack is saved in the alternate stack field in the thread descriptor when the ArbiterProxy switches to the alternate stack. This case occurs when Arbiter A directly or indirectly calls the Arbiter B's target object. When Arbiter B attempts to acquire an alternate stack, it finds that the original stack is saved in the thread descriptor's alternate stack field, and uses the original stack as its alternate stack. Arbiter A still switches back to the original stack and places its own alternate stack in the thread descriptor's alternate stack field when it calls its target object, as described previously; however, Arbiter A has to restore the original stack pointer into the thread descriptor's alternate stack field (in addition to restoring the Arbiter's alternate stack) when the target function returns. The only other change needed for this optimization is that

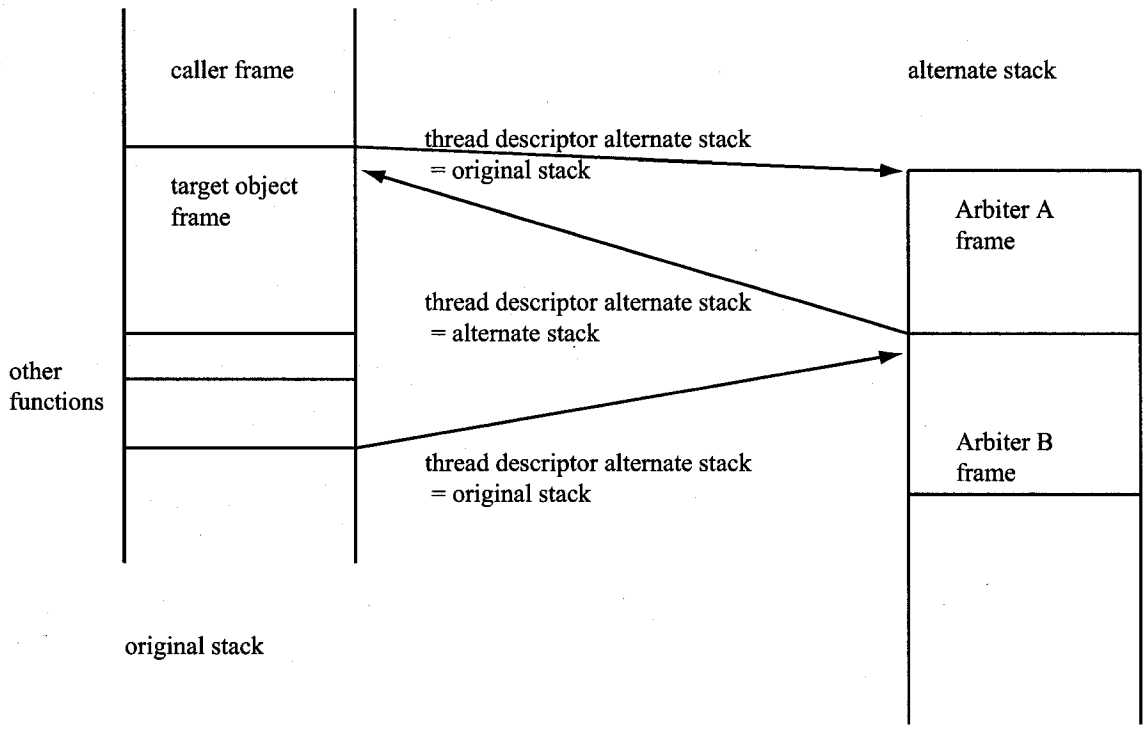


Figure 4.18: Arbiters Sharing an Alternate Stack

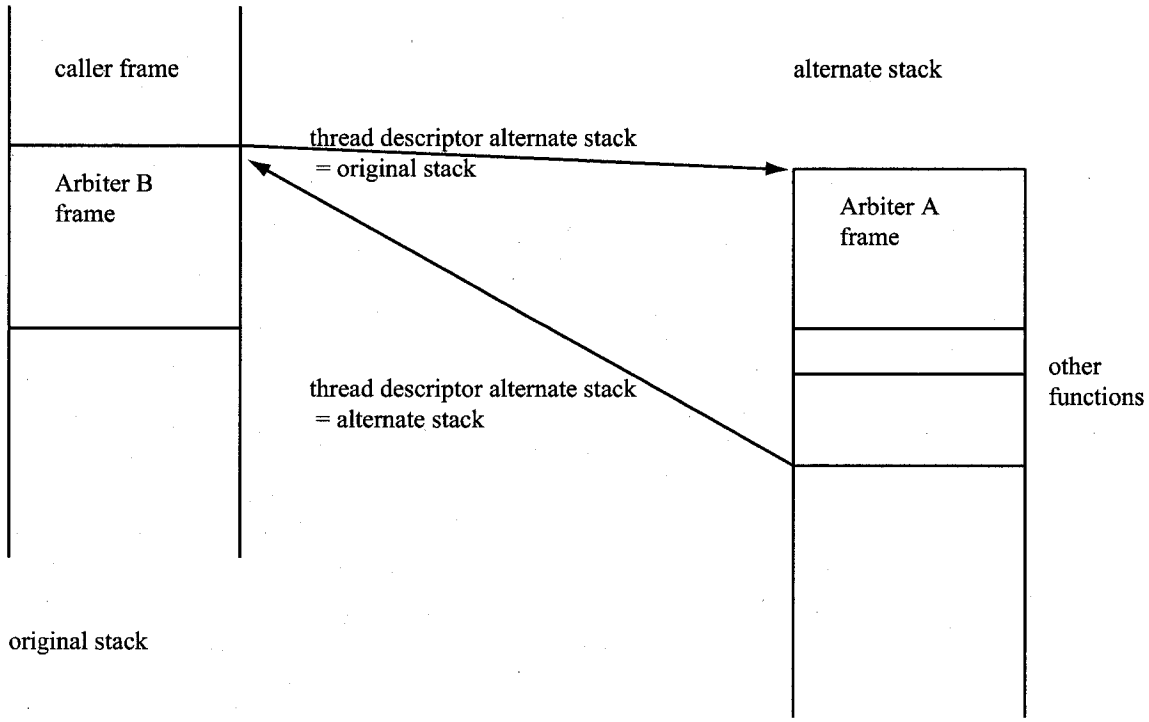


Figure 4.19: Arbiters Sharing the Alternate and Original Stacks

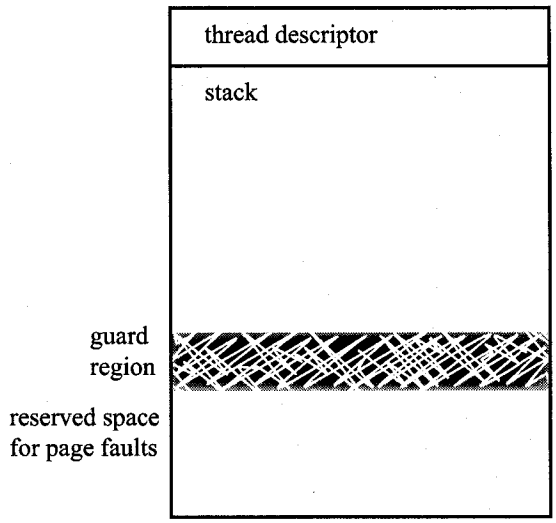


Figure 4.20: Kernel Stack Layout With Debug Check Information Enabled

the ArbiterProxy must clear the alternate stack field when the Arbiter finishes running and returns.

4.4.6 Debug Stack Checks on Page Faults

When K42 is built with debug mode set, the system performs a number of runtime checks that interfere with interposed Arbiters. Specifically, when a kernel thread raises a page fault and K42 is built with debug mode set, it performs some checks to ensure that the current stack is valid and has enough space remaining to handle the page fault because K42 handles the page fault using the active thread's stack when kernel code causes a page fault. Figure 4.20 shows the layout of the stack when debug checks are compiled in, with no Arbiter interposed. At the very top of the stack is the thread descriptor. Below that are the active stack frames, then free space. At the bottom of the stack is a guard area that is filled with a test pattern. The guard area contains a reserved section so that even a full stack will be able to handle page faults properly. The debug code examines the stack pointer to determine how much memory is left on the stack, and inspects the test pattern in the guard region to ensure that the stack is valid and has not overflowed. The end of the page fault handler's stack frame is checked to ensure that the page fault handler will have sufficient space to run in.

When an Arbiter is interposed, there are two problems. First, the check uses the location of the stack relative to the thread descriptor to locate the guard region, but the Arbiter uses an alternate stack that is in a different position relative to the thread descriptor. Second, the Arbiter may raise a page fault while it is switching stacks, and the debug information that is used to verify stack correctness will not be consistent with the state of the stack when Arbiters are interposed.

Two approaches are used to enable debug checks to continue alongside interposed Arbiters. To solve

the first problem, an additional field specifying the end of the reserved section was added to the thread descriptor. When the interposing subsystem switches to an alternate stack, it fills in correct values for the fields in the thread descriptor that are used by the debug checks, specifically the fields pointing to the beginning of the guard area and to the end of the reserved section of the stack. This is an appropriate solution because it allows the debug checks to take place when an Arbiter is running.

Second, debug checks are disabled during the brief periods of time when the information about the stacks is not consistent, and the debug checks were modified to be aware of alternate stacks. Hence, while switching stacks, the Arbiter disables debug stack checks. Disabling the checks reduces their usefulness, so they are only disabled for the short sequence of instructions that manipulates stack pointers. After the alternate stack is in place, the debug checks are re-enabled. This permits the debug checks to take place even when an Arbiter is interposed. Disabling interrupts here would not be helpful, since the primary concern is a page fault caused by the Arbiter switching stacks. Since the debug checks only happen when the K42 kernel is built with debug mode set, the actions the Arbiter takes that are described above also only happen when the K42 kernel is built in debug mode.

4.4.7 Removing an Arbiter

The algorithm for removing an Arbiter is shown in figure 4.21. It considers two cases, one for locating and removing Arbiters in the list of interposed Arbiters, and a special case for removing the Arbiter at the head of the list. The first case applies if another Arbiter was interposed after the Arbiter that is being removed Arbiters are not automatically destroyed when they are removed, since users may wish to access them at a later time to examine any information that they have collected.

Whenever an Arbiter is removed, it examines the global translation table of the target object to determine if its own miss handler is currently installed, or if another Arbiter was interposed after it. In the latter case, the Arbiter knows that an ArbiterProxy belonging to another Arbiter is the miss handler for the target object. If so, it locates the previous Arbiter by recursively calling the ArbiterProxy miss handler's master() function to get the location of the next Arbiter, until it locates the Arbiter that was installed immediately after it was. An example of this is shown in figure 4.11, where there are only two Arbiters. Arbiter A is the Arbiter being removed, and Arbiter B is the Arbiter installed immediately prior to it. Arbiter A acquires its own target lock, and after locating Arbiter B, Arbiter B's target lock as well. After acquiring the locks, it double checks that Arbiter B is still installed (releasing locks and trying again if Arbiter B was removed). Arbiter A then reads its own target and overwrites Arbiter B's target pointer with its own. Arbiter A then directs Arbiter B

```
1  lockTarget()
2  while interposed
3          targetMH = target global translation table entry
4          if targetMH != myArbiterProxyRoot
5                  // another Arbiter is interposed
6                  while targetMH != myArbiterProxyRoot
7                          nextMaster = targetMH->master()
8                          targetMH = nextMaster->target
9                  nextMaster->lockTarget()
10                 // make sure nextMaster wasn't removed while we were getting its lock
11                 if !nextMaster->interposed
13                         nextMaster->unlockTarget()
14                         continue
15                 nextMaster->target = target
16                 nextMaster->resetTargetRepresentativeCaches()
17                 nextMaster->unlockTarget()
18         else
19                 // we are the first (possibly only) Arbiter interposed
20                 lockTargetGlobalTranslationTableEntry()
22                 // make sure no arbiter interposed while we were locking the translation tables
21                 if myArbiterProxyRoot != target global translation table entry
23                         unlockTargetGlobalTranslationTableEntry()
24                         continue
25                 // substitute will install target as the miss handler for targetRef, and put the
26                 // existing GTT entry into dummyMH.
27                 DREFGOBJ(TheCOSMgrRef)->substitute(targetRef, &dummyMH, target)
28                 unlockTargetGlobalTranslationTableEntry()
29         interposed = false
30         unlockTarget()
```

Figure 4.21: Removing an Arbiter

to flush any target representative pointers in its local representatives. To complete the removal, the locks are released.

If the removing Arbiter is installed in the global translation table, it has to restore the global translation table to the original target and reset the local translation table entries. The Arbiter locks its target and the target global translation table entry, and double checks that it is still the first Arbiter installed (releasing the

locks and trying again if it is not) and then sends a message to the other processors in the system to reset the target object's entries in their local translation tables. The removing Arbiter then releases its target lock.

There is a significant issue whereby there may still be threads in flight that have called the Arbiter even after it has been removed. To allow these threads to complete without interruption, the removed Arbiter must remain able to run for some time after the removal. That is, if the Arbiter is destroyed, or if it captures a new target, then it would not be able to properly run threads that remain in flight. To avoid problems, the Arbiter is prevented from capturing a new target, so that any threads that call makeCall() will call the correct target. It is safe to submit the Arbiter for destruction at any time after it has been removed. The K42 garbage collection facility ensures that the object will not be deleted until threads that could have had calls in flight when the destruction request was issued have completed [10].

4.4.8 Changes to K42

The interposition subsystem is modular and does not require many modifications to K42 code that is not part of the subsystem itself. There are three places where changes were required for the implementation. First, the addition of functions to the clustered object system manager to allow Arbiters to change miss handlers in the global translation table and allow the locking of global translation table entries for this change. Due to the manipulations of the global translation table, this was the appropriate place to put this functionality. Second, the addition of the alternate stack field in the thread descriptor. This change was not strictly necessary, since per-thread storage could be implemented within the Arbiter object itself. However, placing this information in the thread offers significant efficiency improvements and allows some of the stack optimizations described previously. The final place where K42 code changes were necessary was the handling of debug checks.

4.4.9 Correctness of Concurrent Operation

K42 is designed to run on multiprocessor computers and as such exposes its components to concurrent operation. Arbiters are exposed to concurrency, from threads running on the same processor or on different processors, in several situations. In these situations, measures have been taken to ensure that correctness is maintained. Arbiters can experience concurrent operation during installation of an Arbiter, during calls intercepted by an Arbiter, and during removal of an Arbiter. Measures are also taken to ensure that no calls can take place during destruction of an Arbiter. Safe concurrent operation of particular functionality added to an Arbiter by the author of specific Arbiters is the responsibility of the author of each particular Arbiter.

Locking in two places protects Arbiters during installation. While installing, Arbiters acquire a lock that protects write access of their target data. The Arbiter must hold this lock to set the target when installing, and also sets a flag that indicates the Arbiter has interposed. This lock and flag prevents concurrent installations of an Arbiter and ensures that an Arbiter can only be installed on one target. In addition, the Arbiter locks the target's global translation table entry to prevent interactions with other Arbiters or other events that would modify the global translation table.

Arbiters begin intercepting calls as soon as its ArbiterProxy object is installed in the global translation table. The installation is ordered so that the pointer to the target's miss handler is stored in the Arbiter before the ArbiterProxy is installed in the global translation table, so the Arbiter is able to correctly locate target representatives as soon as it can receive calls. No resources in the Arbiter are modified while intercepting calls, as the only necessary data is stored on the stack. In the ArbiterProxy, a stack must be acquired by each thread. Stacks are acquired by looking in the thread descriptor (which is different for each thread), and from the stack pool in each ArbiterProxy representative. As described previously, the stack pool is implemented using lock free techniques that allow stacks to be safely acquired during concurrent operation. Hence, calls can be intercepted safely during concurrent operation with installation and other intercepted calls.

Removal of Arbiters uses the algorithm described previously, using the Arbiter target data locks and global translation table entry locks. This ensures correct operation when multiple Arbiters are removed simultaneously. The Arbiter target data lock protects Arbiters from simultaneous removal requests from multiple threads, and also prevents interactions between installation and removal of Arbiters. To ensure that intercepted calls that are in progress proceed correctly while an Arbiter is being removed (and after it is removed, as calls may take longer to complete than the time it takes to remove an Arbiter), it is necessary for the Arbiter to have a valid target, even after it is removed. As long as this is the case, calls can proceed even during and after removal of an Arbiter.

As mentioned previously, the K42 garbage collector ensures that are not destroyed while there are still threads running. Again, the only requirement is that the Arbiter still has a valid target while it is waiting for threads that are still running.

4.5 Specific Arbiter Implementations

To give a better idea of how Arbiters are written and what they can be used for, several specific Arbiters are presented in detail in this section. The ArbiterPassthru is the simplest Arbiter that is able to

emulate the function of the target clustered object by calling the target function. ArbiterBreakpoint and ArbiterCallCounter are used for debugging and performance monitoring, respectively. Finally, a more complex example is presented, namely the Arbiter used for hot-swapping.

ArbiterPassthru interposes on all function calls to a target clustered object. On every invocation it simply calls the corresponding function in the target and returns whatever error code is returned by the target clustered object. This Arbiter was written for evaluating the performance of the interposing subsystem, rather than any practical application. ArbiterPassthru was presented as an example earlier in figure 4.1. The first line gives the class declaration; RepArbiterPassthru is a direct descendant of CObjRepArbiter. RepArbiterPassthru has two functions: handleCall(), which is the function called when a function call to the target clustered object is intercepted, and Create(), which is called to create a new ArbiterPassthru.

A specialization of the (provided) CObjRootArbiterTemplated class is used as the root of the ArbiterPassthru. The class template takes two parameters, the Arbiter's representative class, RepArbiterPassthru in this case, and the representative class for the ArbiterProxy. Here, the basic CObjRepArbiterProxy representative is used, which is appropriate for most Arbiters, including all of the examples given here. The handleCall() function, starting at line 6, merely takes the description of the original call and the index number of the invoked function and passes them to makeCall(). MakeCall() calls the function in the target clustered object and returns the error code resulting from the call to the target object. HandleCall() finishes by returning that same error code to the caller. The Create() function on line 29 is used to create an ArbiterPassthru clustered object. It calls the create function in the root class, and returns the clustered object ID for the newly created ArbiterPassthru object. The root and its ancestor class' Create() functions take care of creating the object's miss handler and installation in the clustered object system, as per clustered object conventions. The code of this very simple Arbiter (figure 4.1) can be used as a template for constructing more complex Arbiters.

The ArbiterBreakpoint shown in figure 4.22 is a simple Arbiter that can be used for debugging. It allows breakpoints to be set at function entry points on a per-object basis. As discussed in Section 5.1, setting breakpoints on a per-object basis is significantly more efficient than setting breakpoints in global code paths, as is done with traditional debuggers.

Structurally, ArbiterBreakpoint is very similar to ArbiterPassthru. There is some of extra code, principally because a custom root object containing an array of flags is used. The flags are used to indicate which functions should trigger a breakpoint. The custom root object takes up lines 4 through 14 of the code. The root's Create() function is typical of clustered object roots. The things to note are the declaration of

```
1  class RepArbiterBreakpoint : public CObjRepArbiter{
2  protected:
3          friend class RootArbiterBreakpoint;
4          class RootArbiterBreakpoint : public CObjRootArbiterTemplated<RepArbiterBreakpoint,
5                                                                        CObjRepArbiterProxy>{
6          public:
7                  DEFINE_LOCALSTRICT_NEW(RootArbiterBreakpoint);
8                  bool breakOn[256];
9                  static Create(){
10                         RootArbiterBreakpoint root = new RootArbiterBreakpoint();
11                         return reinterpret_cast<RepArbiterBreakpoint**>(root->getRef());
12                 }
13                 RootArbiterBreakpoint(){ for(int i = 0; i < 256; breakOn[i++] = 0); }
14         };
15         RepArbiterBreakpoint() {}
16         DEFINE_LOCALSTRICT_NEW(RepArbiterBreakpoint);
17         virtual SysStatus handleCall(CallDescriptor* cd, uval fnum){
18                 if(reinterpret_cast<RootArbiterBreakpoint*>(myRoot)->breakOn[fnum])
19                         breakpoint();
20                 SysStatus rc = makeCall(cd, fnum);
21                 return rc;
22         }
23 public:
24         virtual SysStatus setBreakpoint(unsigned char fnum){
25                 reinterpret_cast<RootArbiterBreakpoint*>(myRoot)->breakOn[fnum] = true;
26                 return 0;
27         }
28         virtual SysStatus unsetBreakpoint(unsigned char fnum){
29                 reinterpret_cast<RootArbiterBreakpoint*>(myRoot)->breakOn[fnum] = false;
30                 return 0;
31         }
32         static RepArbiterBreakpoint** Create(){
33                 return RootArbiterBreakpoint::Create();
34         }
35 };
```

Figure 4.22: Example Code for ArbiterBreakpoint

```
1  class RepArbiterCallCounter : public CObjRepArbiter{
2  protected:
3          uval callCount[256];
4          friend class CObjRootArbiterTemplated<RepArbiterCallCounter, CObjRepArbiterProxy>;
5          RepArbiterCallCounter() { for(int i = 0; i < 256; callCount[i] = 0); }
6          DEFINE_LOCALSTRICT_NEW(RepArbiterCallCounter);
7          virtual SysStatus handleCall(CallDescriptor* cd, uval fnum){
8                  FetchAndAddSignedSynced(&(callCount[fnum]), 1);
9                  SysStatus rc = makeCall(cd, fnum);
10                 return rc;
11         }
12 public:
13         virtual SysStatus getCallCount(unsigned char fnum, uval& count){
14                 *count = 0;
15                 root->lockReps();
16                 CObjRep* rep = 0;
17                 for(void* curr = root->nextRep(0, rep); curr; curr = root->nextRep(curr, rep))
18                         count += reinterpret_cast<RepArbiterCallCounter*>(rep)->callCount[fnum];
19                 root->unlockReps();
20                 return 0;
21         }
22         static RepArbiterCallCounter** Create(){
23                 return RootArbiterCallCounter::Create();
24         }
25 };
```

Figure 4.23: Example Code for ArbiterCallCounter

the array of flags, breakOn, and the initialization in the constructor. The ArbiterBreakpoint representative is similar to that of ArbiterPassthru with two changes: first, there are two additional public functions for changing the status of a breakpoint (lines 24-31), and second, some changes to handleCall() to check if the Arbiter should break on a call or just call it normally (lines 18-20).

Another Arbiter, the ArbiterCallCounter shown in figure 4.23, shows that simple Arbiters can be used for performance monitoring. This example shows how the clustered object nature of Arbiters makes it easier to write efficient data collection tools. ArbiterCallCounter counts the number of calls to each function of the target object. To do this, it uses a distributed array of counters. Each ArbiterCallCounter local representative

```
1   stage = checkStage()
2   if stage = InitialThreadsUncompleted
3           entryCount[currentThreadID]++
4           makeCall()
5           entryCount[currentThreadID]--
6           if stage = InitialThreadsCompleted && allEntryCountsAreZero()
7                   // initial threads completed while we were running
8                   swapTargetObject()
9                   wakeBlockedThreads()
10                  removeArbiter()
11                  makeCall()
12          return
13  else if stage = InitialThreadsCompleted
6           if allEntryCountsAreZero()
8                   swapTargetObject()
9                   wakeBlockedThreads()
10                  removeArbiter()
11                  makeCall()
13          else if entryCount[currentThreadID] > 0
14                  makeCall()
15          else
16                  blockUntilSwapComplete()
17                  makeCall()
```

Figure 4.24: Pseudocode for Hot Swapping Arbiter

has an array, declared on line 3, with a counter for each function in the target object. The array entries are incremented by handleCall() when the function is called on the processor served by that local representative. The atomic increment shown on line 8 is used because there may be multiple threads running on the same processor. ArbiterCallCounter provides the getCallCount() function in its public interface that returns the call counts for a function. This function works by summing the call count values of each local representative. Using this strategy, the cost of obtaining the call count is proportional to the number of representatives, but the cost of incrementing the counter remains constant even if there target object has many representatives in the system. This example demonstrates how the clustered object infrastructure makes this easy to do since Arbiters are themselves clustered objects.

The Arbiter used for hot swapping is more sophisticated than the Arbiters described so far. This Arbiter has not been implemented yet, but the general approach would be to use an Arbiter to intercept calls and allow them to proceed or block them, based on the phase of the switch. Pseudocode is shown in figure 4.24.

The hot swapping Arbiter works in two stages. In the first stage, calls to the target object are monitored, but allowed to proceed. In the second stage, existing threads are allowed to continue until they have left the target object, but new threads are blocked. After all threads running in the target object have drained, the target object is hot swapped, and then the blocked threads are allowed to continue.

Initially, the hot swapping Arbiter monitors calls, but allows all calls to proceed (lines 3-5). The entryCount for all threads is initialized to 0. EntryCount keeps track of how many times each thread is running inside the target object by incrementing each time a thread enters the target object and decrementing each time the thread leaves. Since existing threads may be running in the target object without the Arbiter's knowledge at this stage, it is not safe to hot swap the target, even if the Arbiter is not aware of any threads running in it. Once all of the threads that existed at the time when hot swapping was started have completed, the Arbiter will be aware of all calls that are running in the target object. New calls are then blocked (line 16). To allow recursive calls to complete, the hot swapping Arbiter allows threads that are running in the target object to make new calls into the target (lines 13 and 14). Once the existing calls have drained, the entryCount for all threads reaches 0 (line 7) and it is safe to hot swap the target object (line 8). Once the hot swapping is complete, blocked calls are allowed to continue, and the hot swapping Arbiter is removed (lines 9-12).

In addition to the Arbiter functionality described, the hot swapping Arbiter also needs a way to detect when all of the initially running threads have completed, and needs some method for obtaining new objects and transferring state from the old object to the new one. Thread lifetime detection is already tracked by K42 for use in the garbage collection subsystem, and this information can be used to implement the checkStage() function in line 1 of the hot swapping pseudocode. Methods for creating objects and transferring state must be provided along with the hot swap Arbiter to complete the swapTargetObject() function in line 8.

Chapter 5

Evaluation

The K42 interposition subsystem meets, with only minor caveats, the design requirements set out for it in Section 4.1. In addition, experimental results show that Arbiters introduce only small overheads when they are used. On a 2.0 GHz Apple X-serve G5 running in single processor mode, interposing on a clustered object takes 2404 ns, and removing an Arbiter from a clustered object takes 513 ns. Calling a function that is interposed on by the ArbiterPassthru, as described in Chapter 4, adds 102 ns of overhead. Simple scenarios involving breakpoints and counters using an Arbiter instead of a conditional breakpoint or static counters placed at compile time show that Arbiters are competitive with traditional techniques for debugging and instrumentation when performance is a consideration.

This chapter has two sections. The first examines how the K42 interposition subsystem fulfills the requirements set out in Section 4.1. The second provides an evaluation of Arbiter performance with various microbenchmarks and simple usage scenarios.

5.1 Fulfillment of Requirements

The requirements listed in Section 4.1 are basically met by the current design, as discussed below.

5.1.1 Transparency

Neither the calling code nor the code of the target object needs to be altered to work properly with interposition. Two measures are essential in giving the interposition subsystem this level of transparency. First, Arbiters can intercept function calls with any type signature, allowing Arbiters to be used with functions conforming to the standard K42 calling conventions. Second, alternate stacks are used for Arbiters so that the target object's function runs using the same place on the stack that it would when there is no interposed Arbiter. This allows stack parameters to be passed normally, and means that functions that have knowledge

of the stack they run on will also work properly.

There are, however, a number of ways where an interposed Arbiter can be visible to interposed target objects or their callers. Slightly reduced performance is one visible effect, although the design and implementation aim to minimize overhead. There are also two ways in which an object may see different values in memory if it is being interposed on. The first is in the return address that a function will jump to upon completion. An interposed object will have a return address that lies within a function of the Arbiter as opposed to within the originally calling function. Since an object may be invoked from multiple places, this will typically not be a serious limitation. A second difference is that the translation table entries for the object will be different if it is interposed on. A function in a clustered object normally has a local translation table entry that is equal to its "this" pointer, and a global translation table entry that is equal to the clustered object's miss handler. Since these entries are easy to locate, and the miss handler is typically the root of the clustered object, it would be straightforward for a clustered object to detect that it is interposed on, if it wishes to. This is an unfortunate breach of transparency; however it does not affect programs that are not actively trying to determine if they are interposed on.

5.1.2 Clustered Object Structure

The use of the ArbiterProxy object as the miss handler for the target object enables the Arbiter to be responsible for only one clustered object interface, simplifying the construction of Arbiters by normal programmers. Normal K42 clustered objects are only permitted to have one entry in the clustered object system. With the design presented, the Arbiter is able to provide an interface that other objects can call to control and obtain information from the Arbiter, while the ArbiterProxy intercepts function calls to the Arbiter's target. Although the ArbiterProxy implementation is reasonably complex, it is provided as infrastructure, and typically programmers need not concern themselves with the implementation.

5.1.3 Ease of Use

Writing an Arbiter is easy because of the provided infrastructure, and the fact that the rules that must be followed are not arduous. The Arbiter class hierarchy, shown in figure 4.11, can be extended to allow the construction of simple Arbiters without re-implementing large amounts of functionality. The CObjRootArbiter can often be used as is, and the CObjRepArbiterProxy can almost always be used without modification. Using the ArbiterPassthrough as a template, a simple Arbiter such as the ArbiterCallCounter takes only 12 additional lines of code. It is easy for Arbiters to transparently call their target objects using

the provided makeCall() function. People writing or using Arbiters do not have to worry about low level details of interposition, or about which processors have target object representatives instantiated. Of course, Arbiters that perform complex tasks can be more complex than the simple Arbiters presented so far, and they take correspondingly more effort to write.

There are some simple rules that people writing or using Arbiters do have to follow, however. When calling the target object, Arbiter authors must use the handleCall() function. Users of Arbiters must avoid the recursion that would result if they directly or indirectly called any function of the target object without using handleCall(). Users must also make sure that they do not interpose Arbiters that need services the system is not able to provide. For example, an Arbiter that may cause page faults must not be interposed on a pinned clustered object that runs when the system is not able to handle page faults.

Arbiters are easy to interpose and remove because they provide simple interfaces. The captureTarget() function shown in figure 4.2 is used to interpose an Arbiter on a target clustered object. Its only parameter is the clustered object ID of the target object. The releaseTarget() function shown in figure 4.7 is used to remove an Arbiter from its target. Although Arbiters do not include interfaces for controlling functionality that has been added by an Arbiter author or for retrieving data from the Arbiter, these are easy to add. Because Arbiters are created by inheriting from the Arbiter infrastructure classes provided, new functions can be added to command the Arbiter and retrieve data. These functions can then be accessed through the same clustered object interfaces that are standard throughout K42.

5.1.4 Efficiency

While the performance of the K42 interposition subsystem is fully evaluated in Section 5.2, design choices to achieve efficiency are discussed here. In designing a system, there is a natural tension between efficiency and simplicity (and the attendant ease of use).

The K42 interposing system interposes at a per-object granularity, avoiding global code paths and thus leading to improved efficiency. This is made possible by the indirection through the local translation tables. Figure 5.1 shows how this gives an advantage over systems that interpose on global code paths. The left half of the figure shows globally interposed code that might, for example, occur with trampoline interposition mechanisms. All of the object instances of the same type experience overhead. The right half of the figure shows per-object interposition using Arbiters. If there are two or more object instances of the same class in K42, only one will be interposed on, and the others will not experience any overhead due to the interposition. There is the added overhead of indirection through the translation tables, but this is

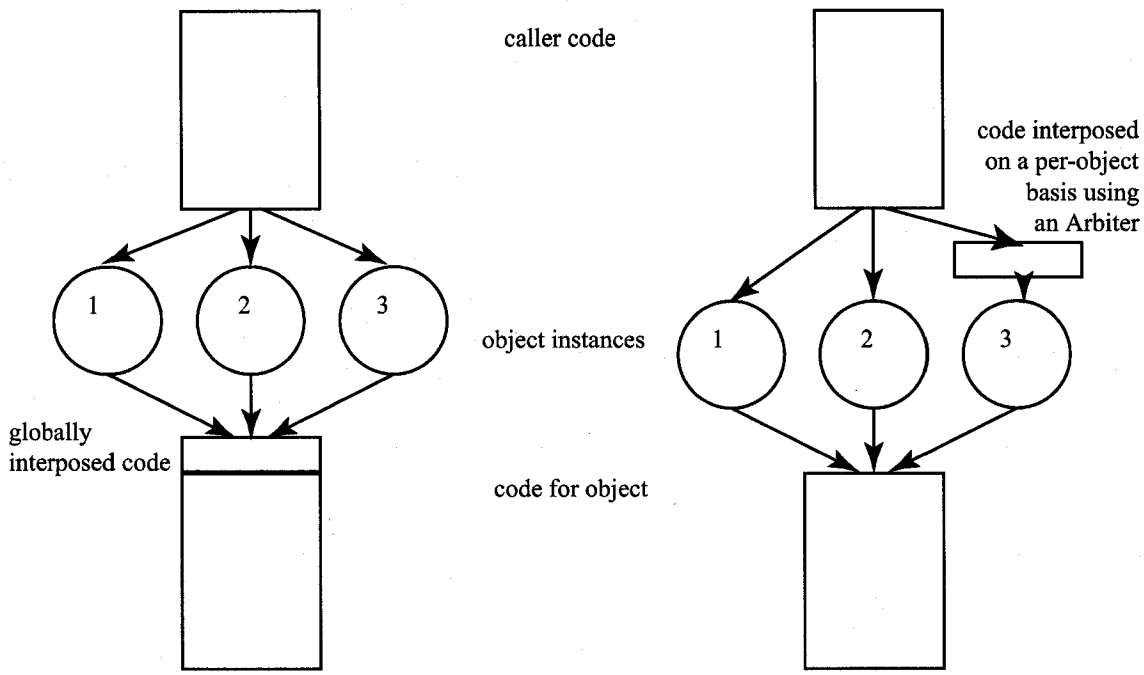


Figure 5.1: Interposing on a per-Object Basis

standard in K42, independent of the interposition subsystem.

The design of the K42 interposing subsystem avoids shared data and locking in common paths to promote efficiency. Shared data reduces performance on multiprocessor systems. Access to shared data increases the likelihood of communication with remote caches, increasing overhead. If shared data is modified by more than one processor then the data may bounce back and forth between the caches of the processors, which can cause significant performance degradation. While the interposition and removal of Arbiters requires setting global data, once a processor has a local Arbiter representative, no global data needs to be accessed to intercept function calls or make calls to target objects. Arbiters do use resources that must be protected from concurrent access on a single processor. Only a single resource, the alternate stacks, must be accessed in the most common case, and it is protected using lock-free data structures, rather than more expensive locks.

Since the code path for interposing is mostly linear, it is relatively easy to optimize. Efficient implementation is a matter of looking at the code that runs and reducing the instruction count, and in particular the number of expensive instructions such as branches and memory writes and reads. Careful attention was paid when writing the Arbiter and ArbiterProxy objects to make sure that the common code paths are short.

Some of the design choices for the Arbiter do lead to unnecessary overhead in order to improve ease

of use, but are designed so that they can be specialized to eliminate the overhead in cases where that is necessary. In particular, the ArbiterProxy object emphasizes ease of use and ease of maintenance. The choice to use clustered object protocols for calling the Arbiter, and the choice to intercept all functions in a generic way causes additional overhead. If there are cases where this overhead is excessive, the ArbiterProxy can be reimplemented, with specialization to make it more efficient for interposition on a particular type of target clustered object. For example, it could be made to bypass interposition on some important function of its target, thereby eliminating overhead for that particular function, and only interpose on the remaining functions. Although this specialization causes significant extra work and hurts maintainability (by linking the specialized ArbiterProxy to a particular target object) this can be done while still taking advantage of the rest of the Arbiter infrastructure.

5.1.5 Generality and Flexibility

Being able to interpose on clustered objects gives the K42 interposing subsystem extensive coverage within the K42 operating system kernel and system libraries. The freedom to run arbitrary code when interposing, including the target function of the intercepted function call, allows interposing to be used for many purposes in K42. This is enhanced by the ability to interact with the intercepted function call by reading and modifying parameters and return values. Arbiters can be written to be general so that they work with all objects (e.g. the ArbiterCallCounter presented earlier), or specialized so that they perform a task that is particular to only one type of object.

To achieve good coverage of the K42 operating system, the interposing subsystem works with clustered objects. With K42's object oriented structure, all system resources and services in the kernel and system libraries are made into components that are implemented as clustered objects. Access to these resources and services is obtained through function calls to the appropriate clustered object. Being able to interpose on clustered objects means the interposing subsystem is able to interpose on accesses to resources and services at a fine granularity. Since interactions between components are necessary for all significant operations that the operating system performs after it has been bootstrapped, the coverage of the interposing subsystem in K42 is excellent.

Arbiters in K42 are able to run arbitrary code, limited only by the state of the system when a function call is intercepted. Arbiters may use resources or services from any clustered object in the system that is available when the Arbiter intercepts a call. Programmers writing Arbiters can call the target function of the intercepted call. They can also block the call or delay it.

As noted in the previous subsection, Arbiters can be specialized to reduce overhead when overhead must be minimized. General purpose Arbiters that can interpose on any object are useful for tasks such as breakpoints and simple performance counters where the type of clustered object being interposed on does not affect the actions of the Arbiter. However, the flexibility to specialize Arbiters to particular clustered object implementations increases the set of possible applications. Arbiters can be used for purposes that require special knowledge about the target object, such as multiplexing between resources or monitoring performance characteristics that are dependent upon the implementation of the clustered object being monitored. Arbiters that are specialized to a particular type of clustered object do not perform any checks when installed, so it is up to the user of the Arbiter to ensure that they are only interposed on appropriate clustered objects.

To help with specialization, the interposing subsystem gives access to the parameters that the function was called with. These can be read and be logged, or used to make calls to other functions. They can also be modified so that an Arbiter can act as a filter that changes values between caller and callee functions. Arbiters can also read and modify the return code from any functions they call, and can view and modify variables that are passed by reference after the function has returned.

5.2 Performance Evaluation

Tests were performed on an Apple X-Serve G5 computer running the K42 operating system. The X-Serve has two PPC970 processors operating at 2.0 GHz. This computer is a modern low-end server. The PPC970 is a high performance 64-bit microprocessor that is derived from IBM's POWER4+ processor. The POWER4+ targets high end workstations and servers with up to 32 processors, while the PPC970 is typically used in single and double processor configuration. The PPC970 features 32 KB of L1 data cache and 64 KB of instruction cache, along with 512 KB of L2 cache. Each test was performed 10 times to warm up the caches and fill in K42's local translation tables, then the test was run 500 times, and the run times averaged. This was repeated 20 times to obtain confidence intervals. For testing purposes, the Arbiters were run inside the K42 kernel, although they are available in user space as well. All tests were run at the "noDeb" debugging level of K42, which is optimized for fast performance, except for the gdb breakpoint test, as breakpoints can not be set in a noDeb kernel using gdb.

5.2.1 Basic Times

Interposing an Arbiter on a clustered object takes 2404 ns, and removing an Arbiter takes 513 ns, as shown in figure 5.2. Interposing and removing Arbiters is significantly more expensive on multiprocessor systems because it requires sending messages to other processors due to the time to create and send messages to the other processors to reset their local translation table entries. Presumably the time would increase proportionally with the number of additional processors in a multiprocessor system. On a large system, adding Arbiters could be a heavyweight operation.

Figure 5.3 shows the time required to call an empty function that has no parameters in the scenarios: (i) without an Arbiter, (ii) using the ArbiterNull that intercepts the call but does not call the originally targeted function, and (iii) using the ArbiterPassthru, described earlier. When compared to a function call to an empty function that has no Arbiter interposed, the basic functionality of the Arbiter is many times slower. The Arbiter takes 94 ns to intercept a function call and return immediately, and 109 ns to intercept a call and call an empty function. That is, it is 13 times slower to have an Arbiter intercept a null function call, and 15 times slower to intercept a null function call and then have the Arbiter call the original target. Section 5.2.4 shows a breakdown of the time spent by the Arbiter.

Although the overhead of using an Arbiter is high relative to calling an empty function, the absolute overhead is acceptable. While the overhead suggests that it would not be feasible to interpose on all clustered objects in an operating system, the remaining results will show that Arbiters can be used in some scenarios without causing performance problems.

5.2.2 Avoiding Synchronization for Stack Acquisition

In some cases, the thread that is calling an object that is interposed on has already acquired an alternate stack. This can happen if a clustered object calls itself recursively, or if a thread's control flow leads it to call multiple objects that are interposed on. When this happens, the Arbiter can avoid synchronization and reuse the stack without the possibility of interference from other threads when allocating the stack. This case is simulated experimentally by having a clustered object that calls an empty function repeatedly, but by interposing on both the calling object and the object with the empty function. Using this setup, the repeated calls to the empty function will already have a stack and will not attempt to acquire another. Figure 5.4 shows the results of this test. Approximately half of the overhead of interposing is due to acquiring a temporary stack. If the stack is already in place, intercepting a function call takes 34 ns, or five times as long as an empty function call, and interposing and calling the empty function takes 51 ns, or seven times

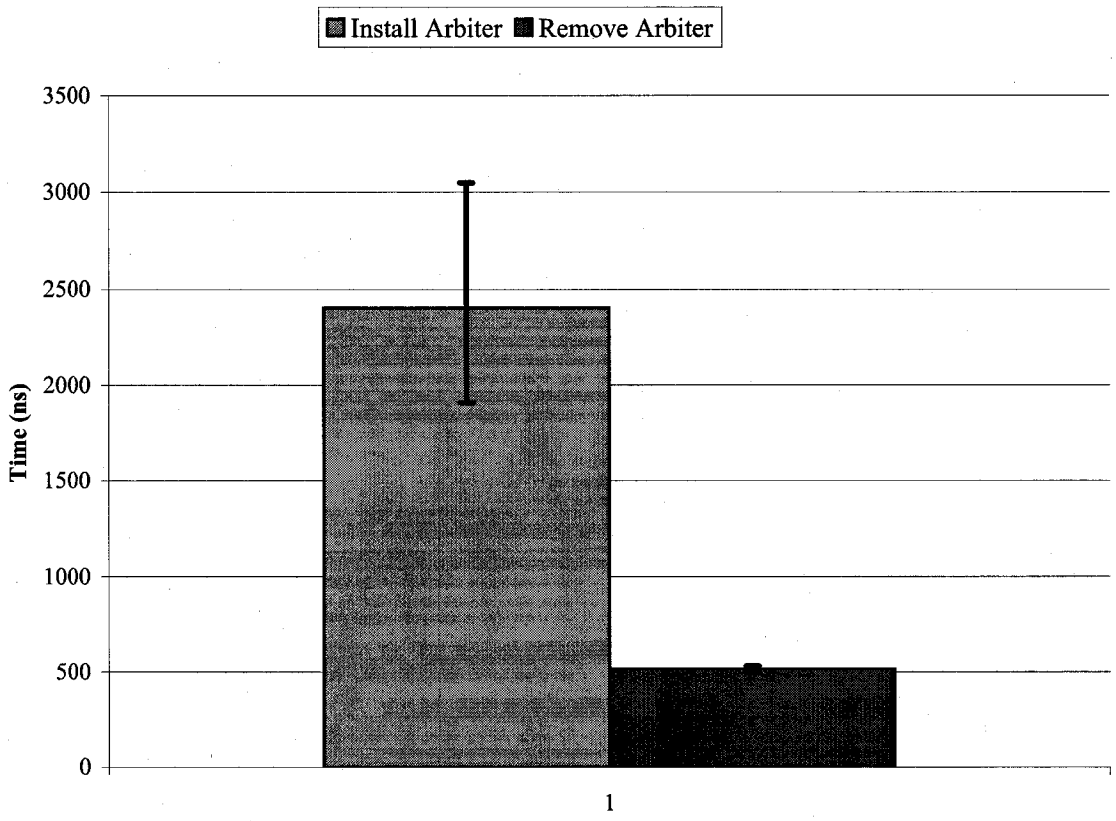


Figure 5.2: Times for Interposing and Removing an Arbiter

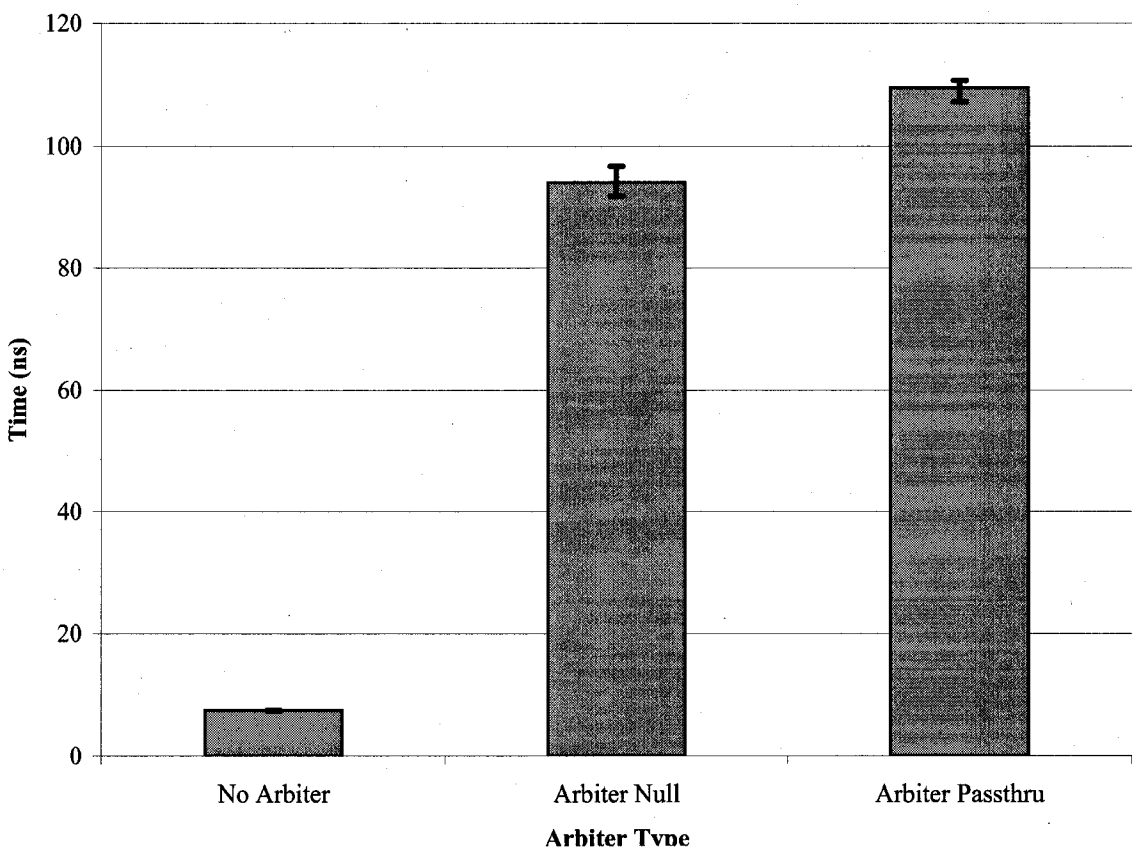


Figure 5.3: Time to Call an Empty Function With Various Arbiters

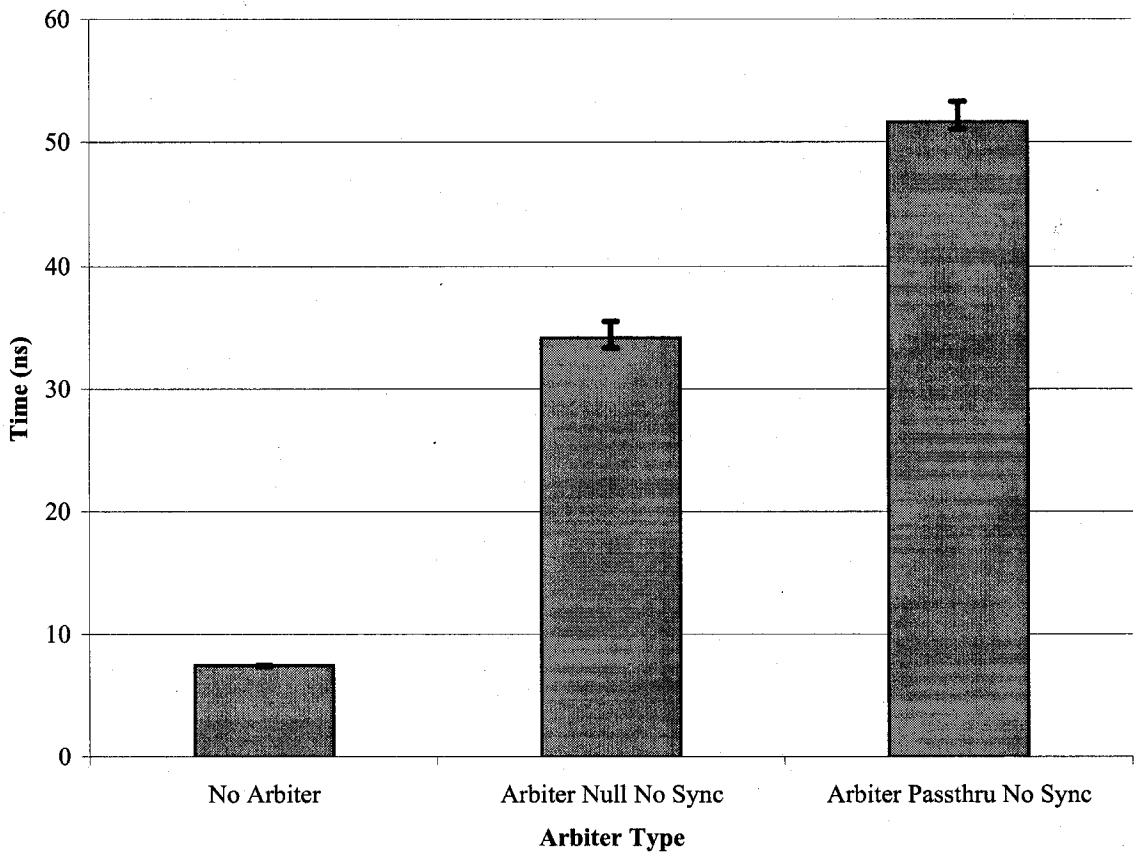


Figure 5.4: Time to Call an Empty Function Without Synchronization for Stack Acquisition

as long as making the empty function call directly. It would be possible to change the design of the arbiter subsystem to eliminate this overhead. However, this change is not obviously beneficial since it would add overhead in other code paths that are more frequently used, such as thread creation. Barring that, it may be advisable to manually install an alternate stack for threads on which heavy Arbiter use is anticipated.

5.2.3 Parameter Saving

Another factor that consumes time in the Arbiter path is the saving and restoring of the call parameters. The Arbiter must save and restore a parameter list consisting of 7 integer parameters and 13 floating point parameters, all passed in registers, each time it intercepts and calls a function. To determine the overhead involved in parameter saving, the Arbiter code was modified so that it did not save parameters. As shown in figure 5.5, Arbiters that did not have to save and load the call parameters took 82 ns to intercept the call, and 89 ns to intercept and then make the original function call. Saving the parameters took approximately 12 ns and restoring them took approximately 8 ns. These numbers are subject to the same absolute error as the other larger numbers, which means that they have a much higher percentage of error than the total call times.

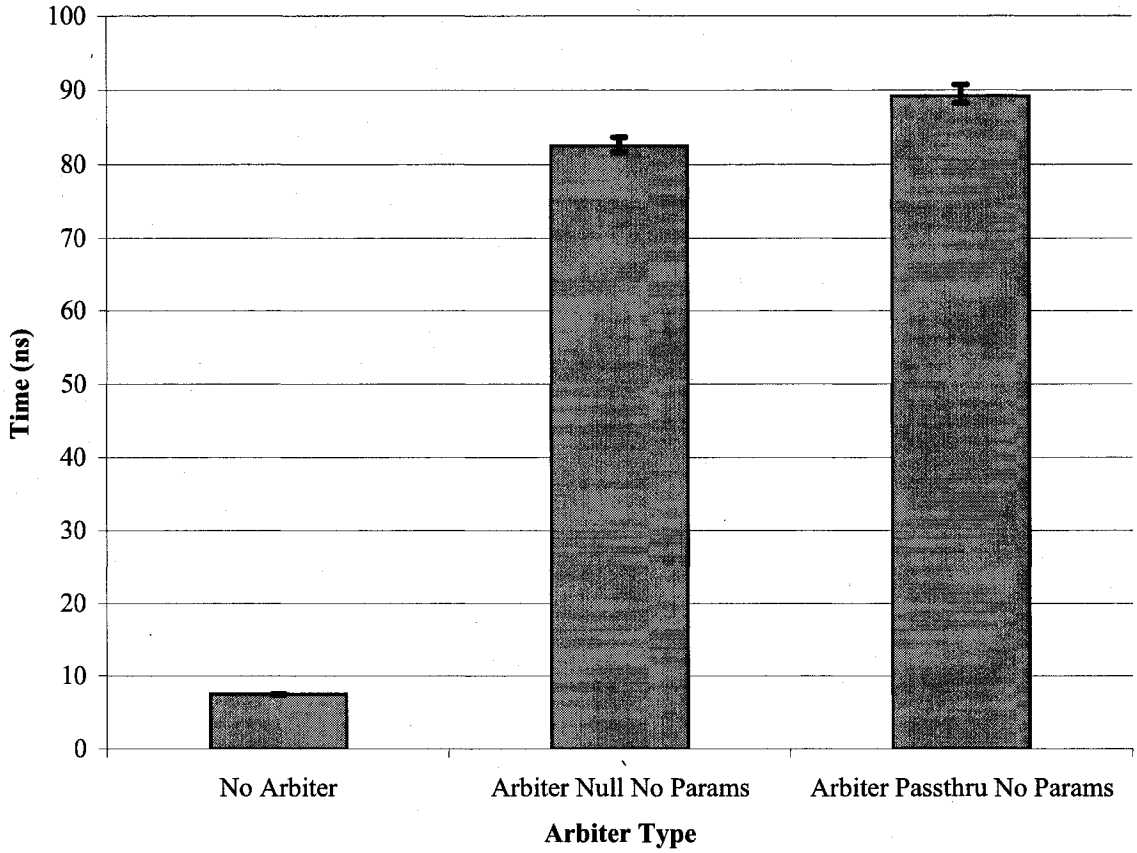


Figure 5.5: Time to Call an Empty Function Without Parameter Saving

The overhead of saving and restoring parameters is small compared to the overhead of using an Arbiter; it is high relative to an empty function call. The overhead related to call parameters could be reduced further by writing a specialized Arbiter version that did not save floating point parameters. Considering the small gain versus the loss in generality, this optimization does not seem worthwhile

5.2.4 Arbiter Overhead Breakdown

From the sequence of experiments performed, the breakdown of the overhead of using an Arbiter was determined to be as shown in figure 5.6. The overhead is given for an ArbiterPassthru intercepting a call to an empty function, and assumes that both the processor caches and the local clustered object representative caches in the local translation table and Arbiter representative are warmed up. The greatest part of the overhead, 53 percent, comes from acquiring and releasing an alternate stack. This is due to the atomic memory access instructions used for synchronization. Calling the empty target function contributes approximately seven percent of the Arbiter's run time. Saving and loading parameters takes up another 19 percent of the overhead of the Arbiter. The remaining 22 percent of the overhead is due to other Arbiter details, including switching stacks, locating representative objects for the Arbiter and ArbiterProxy, and

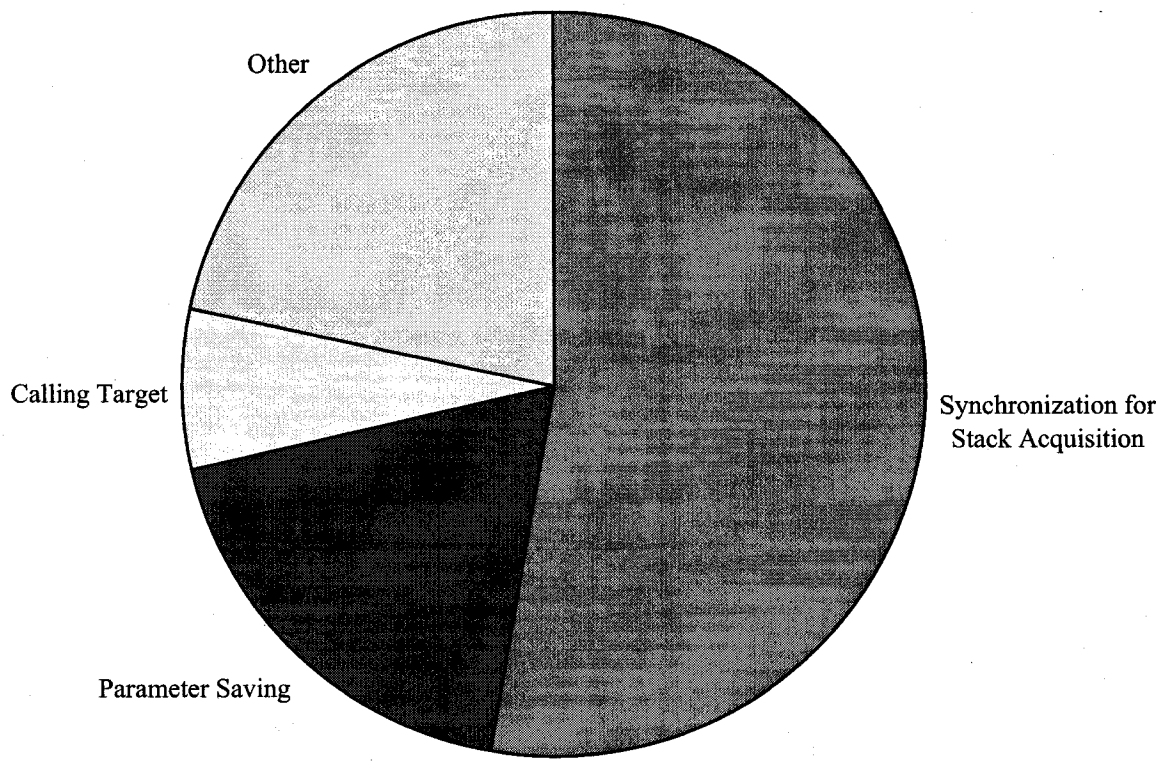


Figure 5.6: Arbiter Overhead Breakdown

calling and returning from the internal functions of the Arbiter and ArbiterProxy.

5.2.5 Performance Overhead of Call Counter and Breakpoint Arbiters

Two experiments were carried out to measure the overhead of two specific Arbiters. The first experiment compares the overhead of the ArbiterCallCounter that was presented in Section 4.5 to the overhead of a counter statically embedded in the program directly. In the second experiment, the overhead of using an Arbiter to implement a conditional breakpoint is compared with the overhead of using gdb to implement a conditional breakpoint.

For the counter test, a function was created that atomically increments a counter each time it is called, and the cost of calling the function was measured. Then, an Arbiter was created that atomically increments a counter each time the object that it is interposing on is called, and the overhead was measured. Figure 5.7 shows the results; calling the empty function with an Arbiter counter interposed took approximately two times as long as calling the counter function; 254 ns compared to 128 ns. Although Arbiters are more expensive than static counters, the difference is not very large in absolute terms. And again, Arbiters can be interposed on specific target object instances, while statically placed counters incur overhead on each

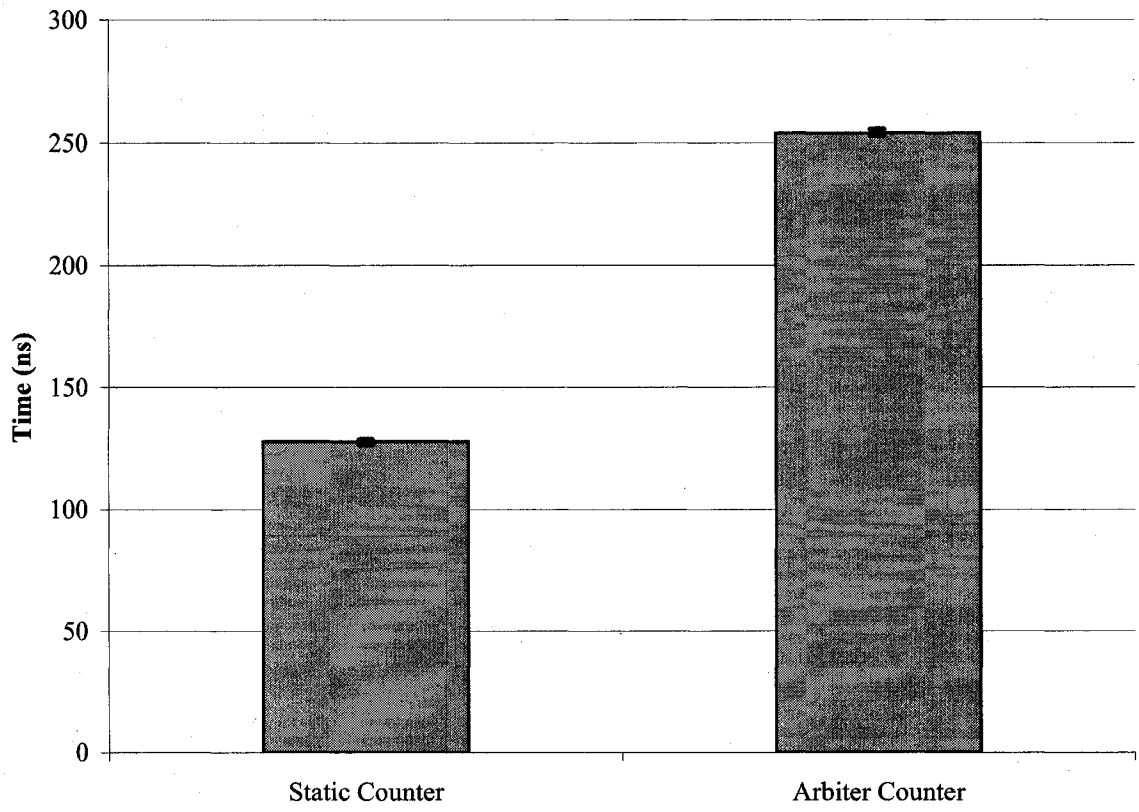


Figure 5.7: Time to Call an Atomically Incremented Counter

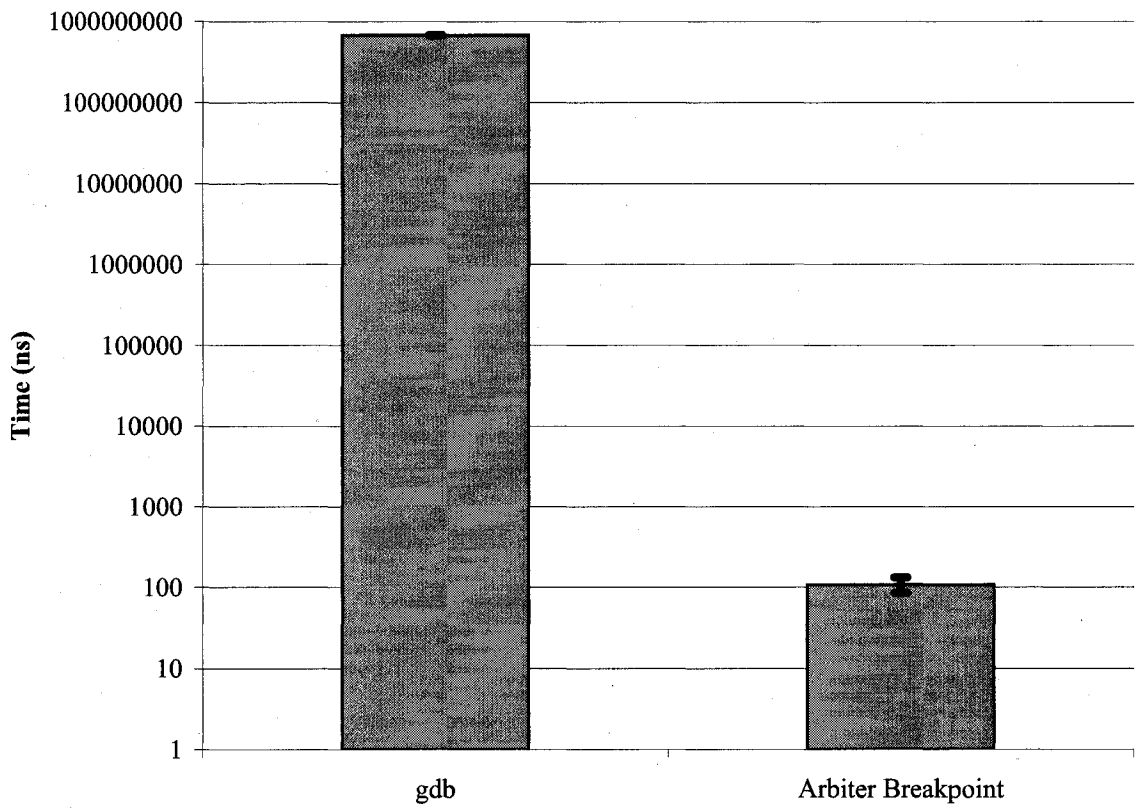


Figure 5.8: Time to Evaluate a Conditional Breakpoint

object instance.

The breakpoint test was carried out by writing an Arbiter that checks a condition each time it is called, and triggers a breakpoint if the condition is true. For the non-Arbiter breakpoint, gdb was attached to the program and a conditional breakpoint was set in the empty function. Breakpoints in K42 are very expensive because K42 does not support running a debugger natively, and the debugger must be attached from a remote machine over the network. A breakpoint condition that always evaluated to false was used, since the goal was to determine how quickly a conditional breakpoint could be evaluated. Figure 5.8 shows the test results; evaluating a breakpoint condition with an Arbiter took 107 ns, whereas evaluating a breakpoint condition using gdb took 677 ms, or more than two thirds of a second. Gdb took more than one million times longer than the Arbiter breakpoint to evaluate a conditional breakpoint. This test is not a fair comparison, however, since conditional breakpoints can be evaluated much quicker than this, but it does represent the situation currently faced by people developing software for K42.

Chapter 6

Concluding Remarks

The primary goal of the interposition subsystem was to allow interposition of code into the kernel and user programs in K42. The interposition subsystem designed and implemented for K42 allows code to be interposed on calls to functions of targeted K42 clustered objects. Code can be inserted dynamically into most parts of the K42 kernel, system libraries and within applications written using K42 clustered objects. The facility implemented supports many applications, including performance monitoring, debugging and hot-swapping. Interposing code allows the behaviour of objects to be monitored, and actions taken each time a particular object is called. Interposition can be used for debugging, either to monitor or act on data or to efficiently add breakpoints or conditional breakpoints at a per-object granularity. Code interposition is essential for implementing hot-swapping, which can in turn be used for on line updates or customization of a running operating system.

The implementation of the interposition subsystem described in this dissertation supports these applications by allowing transparent interposition of code before calls to K42 clustered objects in an easy, flexible and efficient manner. Interposition is implemented by taking advantage of the additional level of indirection through the K42 clustered object translation tables. When a call is made to a function of an object that has code interposed, that function call is redirected through the translation tables to a different object called an Arbiter. The Arbiter can perform arbitrary actions, such as those mentioned previously. Optionally, the Arbiter can also call the original function of the clustered object that it is interposed on. To enable this to be done transparently, the Arbiter is run on an alternate stack, and switches back to the original stack to call the original function.

Arbiters are made easy to use by being written as clustered objects, and by having simple interfaces to initiate interposition on and removal of Arbiters from clustered objects. Similarly, new Arbiters can easily be written, since the basic functionality is provided in the Arbiter base classes that can be easily accessed and extended, since much of the complexity is hidden in an ArbiterProxy object that is provided by the

infrastructure. Flexibility is achieved by allowing interposition on clustered objects within the K42 kernel, system libraries and user programs, and by allowing arbitrary actions to be performed before and after the optional call to the original function.

Careful design and implementation allow Arbiters to be used with low overhead. Per-object interposition avoids overhead on global code paths. The chosen design also avoids locking on the most common paths. Careful implementation reduces the number of instructions that must be executed by an Arbiter, and allows parameters to the original function to be saved and restored once each. The low overhead of the Arbiter is confirmed by experimental results, in particular the 94 ns overhead involved in function calls that are intercepted by an Arbiter.

6.1 Lessons Learned

The work done implementing the interposition subsystem in K42 has shown that code interposition can be done efficiently, and that it has advantages over traditional approaches to debugging and performance monitoring.

Some aspects of the implementation proved more challenging than expected. Obtaining representatives for target objects was challenging. Usually, miss handlers in K42 install their representatives directly into the local translation tables. This is not possible when an Arbiter is interposed, since doing so would bypass the ArbiterProxy and prevent calls from being intercepted. Solving the problem required defining more precisely the relation between miss handlers and translation tables in K42. This had not been done previously, since there had not been any reason to do so.

Another lesson learned was the value of leveraging higher level constructs in lower level code. Machine dependent code was regularly broken throughout the project by changes to higher level code that it relied on. In many cases, these problems could be avoided by referencing higher level stubs that were updated by the compiler each time the system was rebuilt. In other cases, the infrastructure to generate these stubs was not available, because its use is restricted to the K42 kernel and it is not available in the system libraries. In particular, extending the asmConstants infrastructure to the system libraries would have made it easier to implement the K42 interposition subsystem.

The difficulties in dealing with concurrency when removing multiple Arbiters were unforeseen. A significant number of conditions needed to be handled, and multiple locks were required.

Some parts of the interposition subsystem were easier to implement than expected. A good understanding

of how the compiler worked proved invaluable. Switching stacks and calling virtual functions proved to be relatively simple to implement.

The ArbiterProxy was a great help in simplifying the implementation. Separating the Arbiter and ArbiterProxy allowed the clustered object system to treat both the target object and the Arbiter in its normal manner, without modification. If the two parts were contained in only a single object it probably would have been necessary to either modify the clustered object system to treat Arbiters differently than other objects or to make the Arbiter considerably more complex than other clustered objects.

6.2 Future Work

Nevertheless, the experiences so far have been with only a few simple Arbiters. It will be interesting to observe what uses Arbiters will ultimately find given the availability of the interposition infrastructure.

Although the implementation of the interposition subsystem in K42 is complete, there are a few features that may be revisited after sufficient experience is obtained with the use of Arbiters. In particular, the overhead of acquiring an alternate stack for an Arbiter may be reduced significantly (at the expense of additional overhead elsewhere) by allocating an alternate stack on thread creation. The scalability of interposing and removing Arbiters may also need examination if these operations occur very frequently. Although the current scheme works very well for the small SMPs it was tested on, it may become too expensive on computers with many more processors.

Users of Arbiters have already requested the ability to have a single Arbiter interpose on multiple clustered objects simultaneously. Although this is not part of the original design, it is a logical extension.

Arbiters will play an important part in future work on understanding K42 operation and performance. Moreover, Arbiters are already being used for hot swapping in K42. Hot swapping will enable updates and reconfiguration of a running K42 operating system to improve system reliability [6, 7]. Reconfiguration can also be used for dynamic updates to improve system performance in response to changing workloads.

Bibliography

[1] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young. MACH: A new kernel foundation for UNIX development. In *Proc. Summer USENIX*, 1996.

[2] Jonathan Appavoo. Clustered Objects: Initial Design and Evaluation. MSc thesis, Department of Computer Science, University of Toronto, 1998.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, P. Sweeney. Adaptive Optimization in the Jalapeno JVM. In *ACM Conference on Object-Oriented Programming Systems, Langauges, and Applications*, 2000.

[4] M. Auslander, H. Franke, O. Krieger, B. Gamsa, M. Stumm. Customization-Lite. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 43-48, 1997.

[5] R. Azimi, M. Stumm, R. Wisniewski. Online Performance Analysis by Stastical Sampling of Microprocessor Performance Counters. In *Proceedings of the 19th International Conference on Supercomputing*, 2005.

[6] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Jeremy Kerr, Orran Krieger, Robert Wisniewski. Providing Dyanmic Update in an Operating System. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 279-291, 2005.

[7] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert Wisniewski. Improved Operating System Availability With Dynamic Update. In *Workshop on Operating System and Architectural Support for the demand on IT Infrastructure*, pages 21-27, 2004.

[8] Bryan R. Buck, Jeffrey K. Hollingsworth. An API for Runtime Code Patching. In *Journal of High Performance Computing Applications 14 (4)*, 2000.

[9] Bryan M. Cantrill, Michael W. Shapiro, Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pages 15-28, 2004.

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

[11] Kevin Hui. Design and Implementation of K42's Dynamic Clustered Object Switching Mechanism. MSc. Thesis, Department of Computer Science, University of Toronto, 2000.

[12] Randall Hyde. The Art of Assembly Language Programming. *http://webster.cs.ucr.edu/AoA/index.html*, 1996.

[13] Mark Scott Johnson. Some Requirements for Architectural Support of Software Debugging. In *Proceedings of the International Conference on Architectural Support of Programming Languages and Operating Systems*, pages 140-148, 1982.

[14] Michael Blair Jones. Transparently Interposing User Code at the System Interface. In *Proceedings of the Third Workshop on Workstation Operating Systems*, 1992.

[15] Andreas Krall. Efficient JavaVM Just-in-Time Compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 205-212, 1998.

[16] G. H. Kuenning. Precise Interactive Measurement of Operating Systems Kernels. In *Software—Practice and Experience 25*, pages 1-22, 1995.

[17] The Linux Kernel Archives. *http://www.kernel.org*.

[18] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, Maneesh Soni. Read-Copy Update. Ottawa Linux Symposium, 2001.

[19] Przemyslaw Pardyak, Brian Bershad. Dynamic Binding for an Extensible System. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pages 201-212, 1996.

[20] Eric Parsons, Ben Gamsa, Orran Krieger, Michael Stumm. (De-)Clustering Objects for Multiprocessor System Software. *Fourth Workshop on Object Orientation in Operating Systems*, 1995.

[21] Margo Seltzer, Chris Small. Vino: An Integrated Platform for Operating System and Database Research. Technical Report TR-30-94, Harvard University, 1994.

[22] Michael M. Swift, Brian N. Bershad, Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2005.