D R A F T — UUCS-96-004

Microkernels Meet Recursive Virtual Machines

Bryan Ford Mike Hibler Jay Lepreau Patrick Tullmann
Godmar Back Shantanu Goel Steven Clawson

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

`flux@cs.utah.edu`
`http://www.cs.utah.edu/projects/flux/`

May 10, 1996

Abstract

This paper describes a novel approach to providing modular and extensible operating system functionality, and encapsulated environments, based on a synthesis of microkernel and virtual machine concepts. We have developed a *virtualizable architecture* that allows recursive virtual machines (virtual machines running on other virtual machines) to be efficiently implemented, in software, by a microkernel running on generic hardware. A complete virtual machine interface is provided at each level; efficiency derives from needing to implement only *new* functionality at each level.

This infrastructure allows common OS functionality, such as process management, demand paging, fault tolerance, and debugging support, to be provided by cleanly modularized, independent, stackable virtual machine monitors, implemented as ordinary user processes. It can also provide uncommon or unique OS features, including the above features specialized for particular applications' needs, or virtual machines transparently distributed cross-node, or security monitors that allow arbitrary untrusted binaries to be safely executed. Our prototype implementation of this model indicates that it is practical to modularize operating systems this way: some types of virtual machine layers impose almost no overhead at all, while others impose some overhead (typically 10–20%), but only on certain classes of applications.

1 Introduction

Increasing operating system modularity and extensibility without excessively hurting performance is a topic of much ongoing research[2, 13, 30, 35, 6]. Microkernels[18, 1] attempt to decompose operating systems "horizontally," by moving traditional kernel functionality into servers running in user mode. Recursive virtual machines[17], on the other hand, allow operating systems to be decomposed "vertically," by implementing OS functionality in stackable *virtual machine monitors*, each of which exports a virtual machine interface compatible with the "real" machine interface on which they themselves run. Traditionally, virtual machines have been implemented on and export existing hardware architectures so they can support existing "naive" operating systems. (see Figure 1). For example, the most well-known virtual machine system, VM/370[22, 23], provides virtual memory and security between multiple concurrent virtual machines, all exporting the IBM S/370 hardware architecture. However, special *virtualizable (firmware/hardware) architectures*[16, 29] have been proposed, whose design goal is to allow virtual machines to be stacked much more efficiently.

This paper presents a new approach to OS extensibility which combines both microkernel and virtual machine concepts in one system. We have designed a virtualizable architecture and implemented it in software using a microkernel. The microkernel runs on the "raw" hardware platform, which together with a set of higher-level protocols, exports a virtual machine that provides the extended, virtualizable architecture (see Figure 2). Virtual machine monitors (VMMs) executed on this virtual machine can efficiently create additional, recursive virtual machines in which applications or other VMMs can run.

The microkernel's API supports efficient recursion (hierarchical process structuring) in several ways. For memory resources, the virtual machine hierarchy gets explicit support from *relative* memory mapping primitives that allow address spaces to be composed from other address spaces. For CPU resources, the kernel provides a primitive that supports hierarchical scheduling models. To allow

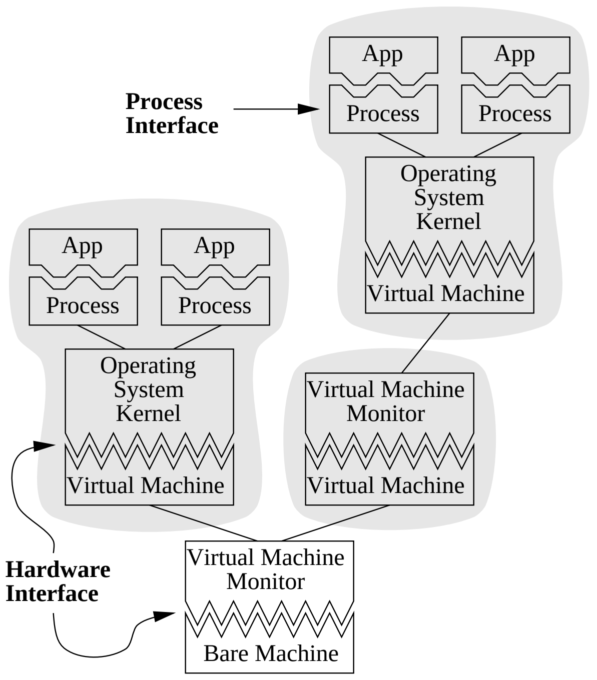

Figure 1: Traditional virtual machines based on hardware architectures

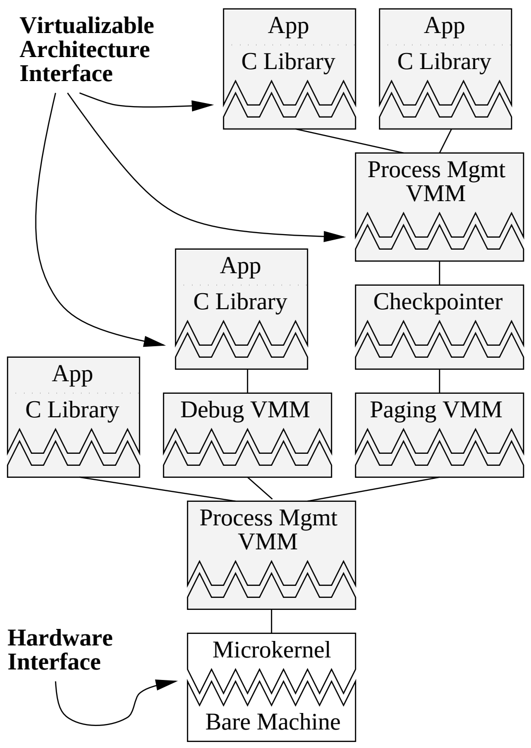

Figure 2: Virtual machines based on an extended architecture implemented by a microkernel

safe short-circuiting of the hierarchy, the kernel provides a global capability model that supports *selective* interposition on communication channels. On top of the microkernel interfaces, well-defined IPC interfaces provide I/O and resource management functionality at a higher level than in traditional virtual machines, more suited to the needs of modern applications: e.g. file handles instead of device I/O registers.

Terminology We now introduce some synonyms, to help reduce awkward and repetitious terms. Henceforth, we will treat the following terms as equivalent: "recursive virtual machine" = "RVM" = "virtual machine" = "VM" = "environment," "virtual machine monitor" = "VMM" = "nester," and "hierarchical" = "recursive" = "nested."

In addition, we will refer to the overall architecture described in this paper as our "model," to avoid confusion with our virtual machine architecture.

1.1 Motivation

Recursive virtual machines can be used to apply existing algorithms and techniques in more flexible ways. Some examples include:

Decomposing the kernel: Some features of traditional operating systems are usually so tightly integrated into the kernel that it is difficult to eliminate them in situations in which they are not needed. The most obvious example is demand paging: although it is often possible to disable it in particular situations on particular regions, (e.g., using POSIX's `mlock()`), all of the paging support is still in the kernel, occupying memory and increasing system overhead. Even systems that support "external pagers," such as Mach, contain considerable paging-related code in the kernel, and most do not allow control over physical memory management, just backing store. The majority of personal computers are dedicated to the use of a single person; hence multi-user security mechanisms are not always needed. A system supporting our model would enable such features to be decomposed into optional virtual machine monitors invoked on a demand basis, and only for the parts of a system for which they are desired.

Application-specific specialization is often desirable; it has been shown that specialized virtual memory management can yield substantially better performance[30, 21, 25] for certain applications which access memory in an unusual fashion (garbage collectors, object stores, relational database systems, some numeric applications). Specialized memory managers are easy to provide in our system.

Increasing the scope of existing mechanisms: There are algorithms and software packages available for common operating systems to provide features such as distributed shared memory[8, 26] (DSM) and security against untrusted applications[43]. However, these systems only directly support applications running in a single logical protection domain. In a recursive virtual machine, any process can create further nested subprocesses which are completely encapsulated within the parent's virtual machine, making them invisible to entities outside the parent. This allows DSM, checkpointing, security, and other mechanisms to be applied just as easily to multi-process applications or even complete operating environments.

Combining OS features: The mechanisms mentioned above are usually designed to work only within the scope of a single application; they are difficult or impossible to *combine* in a flexible manner. One might be able to run an application and checkpoint it, or to run an untrusted application in a secure environment, but existing software mechanisms are insufficient to run a *checkpointed, untrusted* application. A recursive virtual machine architecture allows one to combine these features by layering the mechanisms, since the interface between each "layer" is the same,

Provision of isolated environments, at which today's OSs are very weak. can also be provided by recursive virtual machines. With the advent of Web-based executable content, security issues have become more and more important. Some designs, such as Java, try to achieve security through a combination of language features and runtime verification and control. However, Java requires the use of a special language, and recently uncovered bugs have demonstrated that the implementation of the runtime security mechanisms is error prone.

Virtual machines can provide strong isolation guarantees between subsystems[41], addressing denial-of-service attacks and information leaks through covert channels as well as providing a clean separation between different pieces of mobile code (Applets). Such isolation can also be useful for resource reservation, such as guaranteeing a certain amount of physical memory to real-time applications.

Another example of operating system functionality that we believe could be implemented efficiently as VMMs in this system is a distributed memory manager that provides virtual machines *transparently* distributed over the real machines on a network. I.e., virtualizing local memory to appear distributed to all descendant processes, without their knowledge or cooperation.

1.2 Our Example Virtual Machine Monitors

On top of our software-provided virtualizable architecture we have implemented several traditional operating systems features as independent, stackable virtual machine monitors. These and similar components can be used together in many ways to build highly flexible systems, naturally supporting features that are difficult to implement in conventional operating systems. For example, a checkpointer can be transparently applied to arbitrary domains such as a single application, a multi-process user environment containing a process manager and multiple applications, or even the entire system. In this paper we demonstrate the following specific examples: POSIX process management, demand paging, checkpointing and debugging.

We used micro benchmarks to measure the system's performance in a variety of configurations of the above programs and normal applications. These measurements indicate a slowdown of about 5–30% per virtual machine layer, in contrast to conventional recursive virtual machines, whose slowdown is 20%–1000%[4] Some virtual machine monitors, such as the process manager and debugger, do not need to interpose on performance-critical interfaces such as memory allocation or file I/O, and hence take better advantage of the short-circuit communication facilities provided by the microkernel architecture; these monitors cause almost no slowdown at all. Other monitors, such as the pager and the checkpointer, must interfere more to perform their function, and therefore cause some slowdown; however, even this slowdown is fairly reasonable. These results indicate that, at least for the applications we have tested so far, this combined virtual machine/microkernel model indeed provides a practical method of increasing operating system modularity, flexibility, and power.

1.3 Goals

Our goals in this work are to: (i) Explain and compare our combination of microkernel and recursive virtual machine concepts. (ii) Motivate it by describing useful functionality it can provide. (iii) Elucidate the fundamental kernel properties required to efficiently support such an approach. (iv) Design and implement such a kernel. (v) Establish that none of the required kernel properties is incompatible with a high performance kernel. (vi) Establish IPC-based interfaces and protocols that provide high-level functionality, such as file I/O and memory allocation, in a manner consistent with the recursive virtual machine model. (vii) Exploit the model to demonstrate flexible provision of OS features. (viii) Show that the per-layer nesting cost is moderate and increases only linearly.

The rest of this paper is organized as follows. In Section 2, we compare to related work. We describe our software-provided virtualizable architecture in Section 3. Section 4 describes the properties and design of the kernel we developed to support the virtualizable architecture, while Section 5 describes the high-level mechanisms and protocols we used to implement virtual machine monitors on top of this kernel. Section 7 describes the experiments and results using the four example VMMs. Finally, we conclude with a short reflective summary.

2 Related Work

Most virtual machine systems have, and had, only shallow hierarchies, implementing all functionality in a few layers. In fact, in their heyday, VMs were not driven by modularity issues at all. They were created to make better use of scarce, expensive hardware resources.

Recently, a hypervisor was used to provide fault tolerance (replication) on a whole-machine basis on PA-RISC machines [4]. This application of virtual machines can be approximately compared with our application of RVMs to provide a different form of fault tolerance (checkpointing). This comparison further illustrates the relative merits of each model: The hypervisor approach allows existing operating systems to be run unmodified (HP-UX in this case), but only works on a whole-machine basis (e.g., the same software cannot be used directly for smaller domains, such as a single process or a group of processes.) Of course, this comparison must be taken with a large dose of salt, since the applications and fault tolerance algorithms in question are quite different.

Existing hardware-based VM architectures have several drawbacks: (i) most processor architectures don't support it, since privilege-level information leaks into user-accessible registers; (ii) hardware interfaces are too low-level, making stacking inefficient; (iii) duplication of effort (e.g., double paging), as the whole VMM is duplicated at each level; (iv) there is no way to short-circuit layers and selectively interpose.

Our RVM model has superficial similarities to Unix's hierarchical process organization, in that parent processes can create and control child processes. However, the Unix model falls far short of a true RVM, in at least the following important respects: (i) Parent processes have only a very limited degree of control over their children. For example, they cannot control memory or cpu usage of their children. (ii) Child processes can allocate and use resources that the parent process doesn't own (and possibly never did). (iii) All processes are globally visible in a single process ID namespace. (iv) There are explicit privilege levels. This doesn't mean that the Unix process model isn't useful—in fact, it is very useful, but cannot provide the control needed for the extensibility provided by RVMs.

The Cambridge CAP computer[45] implements a hardware (microcode) architecture that comes fairly close to providing a nested process model. It supports an arbitrarily-deep process hierarchy, in which parent processes can completely virtualize the memory and CPU usage of their child processes, as well as trap and system call handlers for their children. However, the CAP computer enforced the process hierarchy *strictly*, and did not allow communication paths to "short-circuit" the layers as our system does. As noted in retrospect by the designers of the system, this weakness made it impractical for performance reasons to use more than two levels of process hierarchy (corresponding roughly to the "supervisor" and "user" modes of other architectures); thus, the uses of nested processers were never actually explored or tested in this system.

System call emulation and interposition have been used in the past to interpose special software modules between an application and the OS on which it is running. This form of interposition can be used, for example, to trace system calls or change the process's view of the file system[24], or to provide security against an untrusted application[43]. However, these mechanisms can only be applied easily in the scope of a single application process, and generally cannot be used together (i.e., only one interposition module can be used for a given application). Furthermore, although file system access and other system call-based activity can be monitored and virtualized this way, it would be difficult to virtualize other resources such as CPU and memory.

The Exokernel [13] project's work is orthogonal, and possibly complementary, to ours. They're defining where the supervisor boundary is; we don't care where that boundary is, but only about the compositional functionality above it. However, we do care that kernel operations don't have effects on system resources which can't be controlled by VMMs: It's unlikely that the Aegis kernel primitives currently provide the three key properties. Both systems support the ability for applications to have specialized environments, but in ExOS the application binary is modified by linking in OS library code.

Sub-systems supporting stackable and interposable functionality in a particular domain have been an active area of research and development for many years: Jones [24] gives a long list of them. Recent work has benefited from object-oriented structuring, including work on Spring's subcontracts [19] and filesystems [27]. We believe that the careful working out of domain-specific inter-layer protocols is complementary to our RVM work: the high-level component of our VM (the "common protocols") could use those protocols for each class of functionality it provides.

A few existing operating systems, such as KeyKOS[3] and L3[34] have implemented checkpointing on a whole-machine basis in the kernel. While this feature appears practical and useful in some situations, the checkpointing built into these systems is inflexibly tied to the machine boundary: it cannot be used on smaller scopes such as processes or groups of processes, or on larger scopes such as networked clusters of machines. The nested process model allows checkpointing and other algorithms to be implemented over more flexible domains.

Our system borrows many design concepts and abstractions from other systems, suitably modified to support the RVM model, as described in the following sections. For example, our hierarchical memory remapping mechanism has similarities to (and is inspired by) that of L4[35], and appears to provide precisely the "f-map" semantics defined in the recursive virtual machine literature[16, 17], our hierarchical scheduling mechanism is comparable to KeyKOS's *meters*[20], or lottery/stride scheduling's *currencies*[44]. The capability model we use for communication is of course extremely well-known [31]; many of the details of the design and the terminology we use are borrowed from Mach 3.0[11]. The ability to export and re-create all kernel object state appears very similar to the Cache Kernel's [9] abilities in that area. Our kernel object model, in which kernel objects are associated with chunks of user memory, are reminiscent of tagged processor architectures such as System 38[31] and the Intel i960XA. The design of our high-level Unix emulation environment borrows heavily from existing Mach-based multiservers, especially the GNU Hurd[5].

3 "Machine" Architecture

Our virtualizable architecture consists of three components:
First, the extended architecture incorporates only the *unprivileged, "non-sensitive"[17] subset* of an existing instruction set architecture. Limiting the instruction set this way avoids the need to emulate instructions, and makes it possible to implement the virtualizable architecture even on processor architectures such as the PA-RISC, x86, or MIPS, which don't fully support virtual machines based on raw hardware interfaces[4].[1]

Second, a *low-level API* [14] (implemented by the microkernel) provides simple memory management, scheduling, and IPC primitives similar to those of conventional "small" microkernels such as the V++ CacheKernel[9], L3/L4[33, 35], and KeyKOS[20, 3]. This API is designed to support recursive virtual machines efficiently by ensuring that it is not necessary for every virtual machine layer to interpose on and simulate primitive operations such as I/O instructions, page table management, etc. The fundamental properties required to achieve this efficiency are: (a) all primitives are completely *relative*, implying no global resources (e.g., KeyKos's "official" space bank), namespaces (e.g., Unix's PIDs, L4's global thread/task ID's), or privileges (e.g., Unix's root, NT's ACL-based subject security). (b) all state contained in primitive kernel objects (e.g., threads, mappings) is *exportable* as plain data, in a form that ordinary programs can later use to regenerate the objects; and (c) all primitive objects are *owned* by, or associated with, specific virtual machine environments.

Finally, the virtualizable architecture defines the *"common protocols,"* a set of IPC-based interfaces used to implement high-level functionality such as file I/O and memory allocation. In function, these interfaces roughly correspond to the device access conventions in traditional virtual machines and actual hardware, such as the register interface to a SCSI adapter; however, in our architecture these interfaces are much higher-level, closer to the application interface than to the hardware. For example, the primary I/O interface is based on file systems and stream I/O, rather than on bus devices and DMA or programmed I/O. These higher-level IPC-based interfaces eliminate the need to simulate complicated hardware interfaces, and correspondingly simplify and speed up implementations of those interfaces.

[1] We used the Intel x86 architecture for our initial implementation; however, the concepts described here are not processor-specific.

4 The Kernel

The first major component of our OS is Fluke, a microkernel we designed to support recursive virtual machines. Fluke was designed "from scratch" and is an entirely new kernel. Although it is probably possible to implement the necessary support for RVMs in a traditional monolithic kernel, we decided to take a microkernel approach for the proof-of-concept for two main reasons: (i) We felt it would be much more difficult to adapt an existing monolithic kernel, because of the large source base and because the required changes would be widespread. (ii) A monolithic kernel provides much less opportunity to make use of the RVM model. For example, while our checkpointer example would probably still apply, the decomposed process manager and virtual memory manager wouldn't, since these functions are already hard-wired into existing monolithic systems. Of course, because of this decision, our system takes the well-known "microkernel performance hit" due to the additional decomposition and context switching overhead: much more so, in fact, because our microkernel is new and entirely unoptimized. We discuss below that there is nothing about supporting RVM's that is incompatible with a high-performance kernel. In addition, this paper is primarily concerned with showing that *relative* per-layer cost of virtual machine monitors is reasonable, rather than base system performance.

The remainder of this section describes only the aspects of the kernel that are specifically relevant to the RVM model.

4.1 Key Properties

The Fluke kernel does not actually *enforce* a recursive virtual machine model: its API contains no explicit notion of a process hierarchy. However, our kernel API *enables* RVMs by providing a number of vital properties. These abstract properties are described briefly below, and in later sections explored as they are manifest in the Fluke API.

Relativity of kernel abstractions: All kernel objects and abstractions are completely relative: no absolute, global resources or namespaces are made visible through the kernel API. Similarly, there are no special global privileges given to some processes but not others (e.g., no concept of "root"), only privileges of processes relative to each other. Absolute resources cannot easily be virtualized recursively, and therefore would tend to cripple the RVM model. For example, if globally unique identifiers were used to designate kernel objects or communication endpoints, then migrating or restarting checkpointed environments would be difficult because the "unique" identifiers used by the migrated environment on the old system might conflict with identifiers already used in the new system.

Exportability of kernel object state: All kernel objects (e.g., threads, regions) exported through the API are designed so that all of their vital state can be extracted by user-level code and later used to rebuild equivalent objects. For example, this property is obviously crucial for checkpointing to work: otherwise, it would be impossible to save and restore kernel objects used by checkpointed applications, such as threads. However, this property is also required in other cases as well: for example, it enables our out-of-kernel virtual memory manager (MM) and, soon, our distributed memory manager, to demand-page kernel objects as well as ordinary application data.

Object ownership: Finally, our kernel's API is designed so that all kernel objects associated with a particular process can be located and conclusively determined to be "owned by" that process. This property of "ownership" or "process association" is vital to providing control over nested subprocesses to their parents. In just one example of this requirement, without it a process manager has no way to ensure reclamation of all resources consumed by a child process. When its child dies, it needs to be able to track down all the kernel objects used by that process and any descendants it may have spawned. In Mach 3.0, for example, a child task may create new tasks. When the child dies the parent can find a capability to the grandchild, but has no reliable way to determine that the capability actually refers to a task, and assuming it does, whether that task is logically part of the child's state, or was created by some other unrelated task. Also, the child could have simply destroyed its capability to the grandchild, leaving no trace.

The following sections describe in more detail the Fluke kernel primitives and how they provide the fundamental properties listed above.

4.2 Kernel Objects

The Fluke kernel provides only a few types of primitive *kernel objects*, upon which all other functionality is built. *Threads* represent independent flows of control and contain CPU register state, among other things. *Spaces*, *regions*, and *mappings* define the address spaces in which threads execute. *Ports*, *port sets*, and *port references* define communication endpoints. References to non-ports provide handles to most other kinds of kernel objects. *Mutexes* and *condition variables* provide synchronization between threads sharing memory (either within a process or between processes).

All active kernel objects are logically associated with, or "attached to," a small chunk of physical memory. Any process into which a given page of physical memory is mapped can invoke kernel operations on any kernel object in that page, by specifying the *virtual* address of the object within that address space. A thread can create new kernel objects in any memory mapped into its address space that has sufficient permissions; besides the normal `rw` protections, an "object_create" permission must be set. The small user-visible chunk of memory associated with an active kernel object is reserved for the kernel's use. Since this memory can be read and written by untrusted user-level code (even though doing so is a violation of the API), no kernel object state is itself store there; instead, it is used to store hints that speed up the kernel's object lookup upon a system call.

This association of kernel objects with user-level memory provides the notion of object ownership that is needed to support recursive virtual machines. We have reasons unrelated to RVMs—future base performance optimizations—for choosing this design for achieving the ownership property. More traditional descriptor- or handle-based approaches to representing and addressing kernel objects should work as well, as long as the design provides the key properties outlined in Section 4.1.

4.3 Memory Management

Spaces are kernel objects representing address spaces in which threads can execute. Any number of threads can execute in a particular space. One space object is used for each application process, and by higher-level convention, one for each memory segment provided to that process, as explained in Section 6.2.

The actual address space of a Fluke space is defined relative to those of other spaces: it is composed of "views" into other spaces. To manage memory within spaces, Fluke defines two object types: the *region object* which "exports" memory from a space and the *mapping object* which "imports" memory into a space.

A *mapping* object effects "remapping" between spaces, mapping some or all of the address space defined by a region object into another, *destination* space. New regions covering this area in the destination space can be created, allowing the export of that portion of its address space to a third space, and so on. In this way, mappings and regions form a hierarchy of memory sharing relationships. The kernel acts as the *root space*, into which all physical memory is implicitly mapped; it acts as the "ultimate source" of all physical memory.

In order to execute user-level code, the kernel internally "composes" these space-relative mapping and region objects into actual hardware page tables that translate directly from the virtual address space of a particular process into physical memory addresses. This composition mechanism is similar to the *f-maps* described in the recursive virtual machine literature[16].

The kernel's memory remapping mechanism provides the basic "relative memory" support needed to implement nested processes. For example, to create a nested subprocess, a process can simply create a new space object, one or more regions associated with its own space object defining areas of its own virtual address space it is setting aside for the use of the child, and corresponding mapping objects to map these regions into the child space at the appropriate locations.[2] Any threads created in the child process will then execute in that address space, and will only be able to access memory to which it was given access by the parent. The parent can revoke or modify the child's permissions to this memory at any time, allowing the parent to "virtualize" the child's view of memory as desired. Page faults in the child caused by missing permissions are delivered by the kernel to the appropriate parent process.

Note that the kernel provides no primitives for "allocating" memory: storage allocation and management are done purely using high-level protocols. For example, in the situation described above, if the running child process needs more memory (e.g., needs to grow its heap), it must communicate with an ancestor process; the ancestor can then reserve more memory for the child and set up appropriate regions and mappings or grow existing ones as necessary. The high-level protocol for finding and binding to the appropriate memory-serving ancestor process is described later, in Section 5.

Other Hierarchical Memory Management Models: We considered using a design similar to that in L4, which has no explicit abstraction of memory mapping at all; i.e., no "mapping" object. In that model, (physical) memory pages are passed around via IPC messages or by a special kernel operation. Hence, there was no kernel-visible virtual memory hierarchy, just a physical page hierarchy.

Although this model may have worked for those managers that did not want complete control over memory they handed out, it made it extremely hard for those that did. Specifically, this would not allow DSM to be implemented transparently over multiple processes by a user-mode process. A process several levels removed from a DSM manager might flush page mappings from its children, and the manager would never know. If there is an explicit object representing an area of memory, it provides a handle for detecting such cases.

4.4 Interprocess Communication

IPC in Fluke is based on a capability model similar to that of Mach 3.0. A *port* provides the server endpoint of a communication channel, while a *port reference* provides the client-side endpoint. A Fluke message consists of a stream of raw, uninterpreted bytes, plus an optional sequence of port references (capabilities).

[2]The actual method of creating nested child processes in our system, described later, is a little more complicated in order to provide greater flexibility; however, the simple method described here works fine and illustrates the basic concept.

The capability model used in Fluke supports recursive virtual machines in a number of ways. First, it provides the notion of relativity in the communication mechanism essential to the nested process model: given a capability to a file service, for example, the client need not know where or how the file server is implemented, or what intermediaries, if any, may be interposed on the communication channel. Since a parent process that creates a nested subprocess controls what capabilities it initially gives to the subprocess, it ultimately controls all communication across the "boundary" containing the nested subprocess. If the nested subprocess creates further subprocesses, resulting in a full nested environment, then the processes in this environment can freely communicate among themselves with no interference from or knowledge by the parent; however, communication with entities outside of the environment can still be controlled by the parent as desired.

Since these are microkernel-mediated capabilities and therefore not directly accessible to any user process, they can be passed freely between RVM layers, without compromising anyone's security. This contrasts with the Cambridge CAP computer[45], for example, in which capabilities could not be passed between process hierarchy layers because the bits representing a capability in one process are directly accessible to the code running in its parent process. The ability of capabilities to be passed arbitrarily between our RVM layers allows communication to short-circuit the layers in many cases, as described later; this property is very important for maintaining good performance, because it allows parent processes to interpose *selectively* on IPC channels entering or leaving the subprocess, rather than being forced to interpose on *all* IPC, which would result in a much larger performance penalty.

Even though a parent process does not have direct access to the raw bits describing capabilities in its nested child processes, The Fluke API allows a parent to determine if a given capability refers to an object under its domain of control, and if so, which one. For example, our checkpointer uses this functionality to detect and "passivate" capabilities in one part of the checkpointed environment that refer to other objects elsewhere in the checkpointed environment, so that these objects and capabilities can be transparently restored on restart. Capabilities referring to objects outside of the checkpointed environment will not be "recognized" this way and must be handled separately; these issues are discussed later in Section 6.3.

In providing the "exportability property," the deterministic and synchronous Fluke IPC semantics are also relevant. Fluke IPC has no message queues, avoiding the problem or impossibility of retrieving messages in such an intermediate state. If Fluke IPC blocks, e.g., due to a page fault, the thread state, buffer offset, and residual length are rolled-back to a point at the kernel entry boundary. This aids in the provision of simple exportable semantics.

4.5 Scheduling

The final type of resource the Fluke kernel directly deals with is CPU time. As with memory and communication, the kernel provides only minimal, completely relative scheduling facilities. Threads can act as schedulers for other threads, donating their CPU time to those threads according to some high-level scheduling policy; those threads can then further subdivide CPU time among still other threads, etc., forming a *scheduling hierarchy*. The scheduling hierarchy normally "follows" the virtual machine hierarchy, in a loose sense, but is not required to. The higher-level "common protocols" determine the actual scheduling hierarchy.

The details of scheduling under Fluke [15] are beyond the scope of this paper; only its relative, hierarchical nature is important to the RVM model. Other hierarchical schedulers, such as the *meter* system in KeyKos[20], and lottery/stride scheduling[44], should also work in our RVM model.

4.6 Security

The Fluke kernel currently contains no special security mechanisms; all low-level support for security is integrated into the other primitives exported by the kernel. Memory access security is provided by the memory mapping and protection mechanism, communication security is provided by the capability model, and CPU usage security is provided by the hierarchical scheduling mechanism.

This suffices for many environments. However, to satisfy the most demanding security-assurance needs such as the most stringent of the TCSEC[39] classes, it appears important to provide explicit support for traditional subject-based security. We are working with others who are adding such support to Fluke. Our intent is to provide a means to virtualize the ensuing security identifiers, preserving the "relativistic property" of the interface. We are evaluating whether this "security enforcement" can and should be implemented by an ordinary process or at the kernel level, as is traditionally done.

It is worth noting that some other kernel-level security models are likely also to be compatible with the RVM model, such as the Clan/Chief model used in L3 [32], or the hierarchical subject-based security model used in VSTa[42].

5 High-level Protocols

In order to demonstrate how our model can be applied to "real" systems, we have implemented a partial POSIX environment on top of the Fluke kernel, using VMMs to provide traditional Unix kernel features, such as process management and demand-paged memory, although in a more flexible way.

5.1 Common Protocols

A crucial component of our virtualizable machine architecture is the *common protocols*: a set of standardized interfaces used to communicate between VM layers. Whereas the underlying Fluke IPC mechanism provides primitive I/O channels, comparable to I/O ports in hardware-based virtual machine architectures, the common protocols define the communication protocols used on those ports, analogous to the register programming conventions used to program hardware devices.

There could be more than one set of common protocols which define distinct virtualized architectures; in this section we consider the common protocol suite used to implement a partial POSIX environment on top of the Fluke kernel. While many of the protocols are designed specifically for POSIX (e.g., the process management interface) some are more general (e.g., the memory management and file I/O interfaces) and could be applicable to other environments. The POSIX common protocols, hereafter referred to as "the Common Protocols" or CP, are a set of hierarchically structured interfaces defined in CORBA IDL.

Parent interface. This is the top level interface used for parent/child communication, which effectively acts as a "name service" interface through which the child requests access to other services. This is the only interface that *all* VMMs interpose on; a VMM selectively interposes on other interfaces only as necessary to perform its function. The overhead of this interposition is minimal because typically only a few requests are made on this interface, during the child's initialization phase, to find other interfaces of interest. The parent interface currently provides methods to obtain initial file descriptors (e.g., `stdin`, `stdout`, `stderr`); find a filesystem manager, find a memory manager, find a process manager, and exit.

Filesystem interface. The file system interface in our system is similar to those of other microkernel-based operating systems that support independent file servers, such as Spring[27] and the GNU Hurd[5]. It provides methods closely corresponding to POSIX file I/O calls, such as `open`, `link`, `unlink`, `rename`, `mkdir`, etc.

Memory Management interface. The Common Protocols memory interface exports memory *segment* and *pool* abstractions. A memory segment represents an arbitrary-size chunk of allocated memory which can be mapped into a process's address space. The segment interface includes methods allowing clients to map segments, change the size of variable length segments, destroy segments, etc. When a segment is destroyed, all Fluke objects in its memory are destroyed and the segment's memory pages freed. A *memory pool* is a collection of segments and other (sub) pools used to account for and reclaim "anonymous" memory. Memory pools provide methods to create and destroy sub-pools, and to allocate segments from the pool. Destroying a pool destroys all segments allocated from it and, recursively, all sub-pools derived from it. In short, segments represent actual memory while pools provide a convenient mechanism for resource control and accounting.

Process Management interface. The process management interface supports POSIX process-related functionality, such as `fork`, `exec`, `getpid`, etc. It also provides the means for processes to send POSIX signals to each other.

5.2 Libraries

In our system, most of the POSIX functions that are traditionally implemented as system calls are actually implemented by the C library residing in the same address space as the application using it. These C library functions then communicate with parent VMMs and external servers as necessary to provide the required functionality. For example, each process's file descriptor table and its current directory are tracked in the process itself, as Fluke IPC capabilities referring to file servers. The file descriptor table itself is managed purely by the local C library.[3] Our C library supports multithreaded applications and servers by providing a subset of the POSIX.1b threads interface ("pthreads").

Whereas the common protocols can be considered part of the machine architecture in that they must be supported at each virtual machine interface in order to provide stackability, the C library is purely internal to VMMs and application processes; VMMs and applications could be written using completely different libraries without affecting compatibility or VMM stackability. The C library in our system is somewhat comparable to IBM's Conversational Monitor System (CMS), a minimal single-application "operating system" designed *only* to run under virtual machines, which provides high-level services as a convenience to applications without actually implementing significant OS functionality itself.

The Nesting Library. The *nesting library*, generally linked only into virtual machine monitors and not ordinary servers or applications, provides the "parent-side" complement to the C library: it provides basic facilities to support applications that create nested subprocesses. For example, it contains standard functions to spawn a nested subprocess given an arbitrary executable file image. Use of this library is again completely optional: applications can always create nested virtual machines manually in whatever way they desire; this library only provides a "standard" mechanism for creating child virtual machines and providing Common Protocols-compatible interfaces to them.

Although these libraries are currently statically linked, once we implement shared libraries in our system, it will be possible to share this library code even across VM layers. This is because the Fluke relative memory mapping mechanism is not constrained to follow the virtual machine hierarchy strictly: mapped file images can be exported from an arbitrary file system server directly into any task that can access the server (i.e., has a capability referencing the server with sufficient permissions).

[3] However, the actual files and "open file descriptions", containing seek pointers and most other per-open state, are maintained by separate file server processes; this greatly simplifies some of the traditionally hairy "multiserver issues."

5.3 Bootstrapping: the Kernel Server

Besides implementing the basic microkernel API used by all virtual machine layers, the Fluke kernel also implements a minimal Common Protocols interface defining the environment presented to the *first* user-level application loaded directly on top of the kernel (the "root" virtual machine). This initial CP interface consists of a physical memory interface and a minimal root file system interface.

The memory pools exported by the kernel provide the full memory pool interface defined by the Common Protocols; however, memory segments allocated from the kernel's pools always refer to unpageable physical memory. If demand paging is desired, an appropriate virtual machine monitor must be loaded on top of the kernel.

The kernel's root file system interface exports a simple memory-based file system whose initial contents are a set of *boot modules* loaded into physical memory by the boot loader along with the kernel. These files typically contain executable images for VMMs and other components that must be loaded before a "real" file system. The minimal root file system supports file creation, reading, writing, etc.; however, as with the root memory pools, all files on this root file systems are stored in unpageable physical memory. If persistent, disk-based files are needed, then an external file system must be run on top of the kernel; the kernel's root file system can then be destroyed in order to free up physical memory occupied by the initial files.

The kernel does *not* provide any process management interface at all; therefore, in order to run applications such as shells which create and manipulate POSIX processes, a process manager must be run on top of the kernel.

6 Example Virtual Machine Monitors

We now detail the user-level applications that take advantage of the model to provide OS features in a more flexible way.

In the following sections we describe these examples which we have implemented: POSIX process management, demand paging, checkpointing, debugging, and tracing. We also outline an unimplemented example: a distributed memory manager (DMM) cooperating with other DMMs through IPC to create one large transparently distributed environment out of several independent environments.

6.1 The Process Manager

We implemented a virtual machine monitor that creates a POSIX-like multiple-process environment, with each "process" being a separate virtual machine implemented by the process manager. The process manager keeps track of process IDs, handles interprocess signals, `fork()` and `exec()`, and implements other high-level mechanisms expected in a Unix-like environment, as defined in the `Process::` Common Protocol. The process manager is a completely optional component: applications that don't `fork()`, send signals, etc., can be run without one. Furthermore, unlike even in most microkernel-based systems, multiple process managers can be run side-by-side or even arbitrarily "stacked" on top of each other to provide multiple independent POSIX environments on a single machine.

The process manager's basic function is to allow multiple *peer* processes to coexist at the same nesting level and interact with each other as processes do in traditional systems. The other nesting modules we implemented can only run a single nested subprocess at once; "spreading" the tree is left to the process manager (PM).

The PM communicates with its child processes by intercepting messages on their process port. It should be pointed out, however, that the processes can and do directly use the facilities provided by the Fluke kernel API. For instance, the `fork()` operation only registers a new process with the PM. Creating new memory segments, copying the memory segments, copying the kernel objects, and starting the necessary threads in the child process are all done directly by the parent task.

The PM does not maintain memory. Instead, when queried for its `MemPool` interface it passes on the `MemPool` port reference obtained from *its* parent, referring the tasks it manages to whatever memory manager it happens to run under. This can be the kernel server in a realtime system which uses physical memory only, or a virtual memory manager at any point in the nesting hierarchy. Future requests are sent directly to that memory manager.

Multiple Process Managers
Some microkernel-based OSs, such as Mach, have been able to run multiple independent high-level operating environments simultaneously by running multiple instances of the necessary servers. However, doing so generally required that the "nested" servers be somewhat modified (e.g., `#ifdef`'d) in order to conform to the interfaces exported by the previously loaded operating environment rather than those exported by the "raw" microkernel. Also, once launched, it was often difficult for the parent environment to control the child environment: for example, to control the amount of memory it uses, or to find and kill all the processes it may have created if the sub-environment is to be terminated. These were problems in all of the existing Mach-based servers, for example, such as UX, Lites, and the Hurd. Under the nested process model, these problems do not arise.

6.2 The Virtual Memory Manager

We implemented a user-level demand paged virtual memory manager that creates a virtual machine whose anonymous memory is paged to a swap file. Arbitrary programs can be run in this paged virtual machine, such as a single application, or a process manager supporting an entire paged POSIX environment similar to a traditional Unix system. Since demand paging is implemented as a separate component instead of being lumped with other features such as multiuser security, it is much easier to avoid problems with traditional virtual machine monitors related to duplication of effort, such as double paging[17, 37].

Our prototype memory manager (MM) is implemented as an ordinary user-space application program, which loads and runs another application program (specified on the MM's command line) in a virtual memory environment. The memory manager implements the complete Common Protocols memory interface, while "passing through" the interfaces such as file systems and process management, with no interposition. The MM provides anonymous memory segments backed by a swap file and cached in its own address space.

On startup the MM obtains a memory segment of a specified size from its own memory manager. This segment is the physical memory that the MM virtualizes.[4] The MM then spawns the application to be run, interposing on its Common Protocols parent interface. The manager passes on (via its parent port) all requests on that port except for the request for a memory pool, which it provides.

Creation of pools and sub-pools involves allocation of a new object and port reference to return to the caller. When a memory pool segment creation request is made, the MM allocates the necessary address space resources. In addition to providing memory pages, the MM must be able to return a reference with which a client can map the segment with a given protection and it must be able to handle page faults that occur within the segment. The Fluke region object provides these capabilities. Use of a region requires that a segment occupy a contiguous range of memory. The memory manager accomplishes this by creating a separate Fluke space object whose only function is to provide address space for the segment's region and memory. The manager maps ranges of physical memory into this space as required.

When physical memory is freed, either because of explicit segment destruction or because of page replacement, the MM must deal with any Fluke kernel objects that were present in the memory. Using a Fluke microkernel call, the manager locates all objects in the affected range of memory. In the case of segment destruction, it can then just destroy the objects. However, for page replacement the ob-

[4] Though this segment may in fact be virtualized by a previously-loaded memory manager, we refer to it throughout this section as the "physical" memory that the memory manager provides.

jects need to be preserved and later restored when the memory is paged in. The simplest approach for doing this is to move objects (using a Fluke kernel call) into MM private memory at pageout time and to move them back at pagein time. A more sophisticated method takes advantage of the microkernel's ability to completely export kernel object state to page objects out along with "raw" memory just as the checkpointer does.

We have deployed the MM in two configurations. In the simplest configuration, the MM does no paging and is virtualizing memory only in the sense of naming (i.e., remapping virtual addresses). Here, requests for new memory segments are fulfilled by allocating the appropriate amount of physical memory at segment creation time. Thus the application environment can only allocate as much virtual memory as the MM has physical memory. Also, since no page replacement is performed, the MM only implements destruction of Fluke objects.

In a more conventional configuration, the MM allocates a Fluke space and region at segment creation time but physical memory is allocated on demand. When a segment page is first referenced, a fault is generated which is directed to the Fluke region's "keeper" port which is held by the MM. The MM can then allocate physical memory, map it into the host space at the appropriate location, and return to the application to retry. During page replacement, the MM currently just moves objects into its memory.

The MM is free to implement whatever page replacement policy it chooses. This could be an internal global policy for its physical memory pool, or segment-specific policies negotiated with applications through a higher-level protocol.

6.3 The Checkpointer

We implemented a user-level checkpointer that, like the demand pager, can operate over a single application or an arbitrary environment, transparently to the target. By loading a checkpointer in the "root" virtual machine immediately on top of the microkernel, a whole-machine checkpointed system can be created similar to that provided in the kernel by KeyKOS[28] and L3[33]. To our knowledge this is the first checkpointer that can operate over arbitrary domains in this way.

Checkpointing Algorithm
Our checkpointer currently uses a simplistic sequential checkpointing algorithm: to take a checkpoint, it stops all the threads in the child process, saves the contents of the child's memory (including the state of any kernel objects the child process has created in its memory), and then reenables the threads to allow the child process(es) to continue execution.

This algorithm, of course, will not scale well to large checkpointed applications or environments, or to distributed environments. However, more efficient single-process checkpointers based on well-known algorithms [12, 10] could also be implemented in our environment, in the same way.

Checkpointing memory
Because the checkpointer interposes on the memory allocation interface, it has specific knowledge of what memory the application has asked for and what memory it is using. This direct access is also used to find kernel objects: using a Fluke microkernel call, the checkpointer locates all objects in the relevant regions.

Checkpointing kernel objects
There are two classes of kernel objects that a Fluke checkpointer must deal with. First are those objects created within the child environment which only reference kernel objects internal to that environment. To preserve the state of these objects we create unique id's for each object and represent inter-object references with these id's.

The second class of objects are those with references to kernel objects outside of the scope of the checkpointer, for example a reference to the memory server, or open files. Any external reference owned by the child environment must have been granted to it by its parent. For example, memory mappings in the child environment will contain references to the exported regions in the checkpointer. These references will be flagged as exported region references, and replaced with equivalents at restart.

A checkpointer can choose to interpose on as many potential external references as it likes. Our implementation chooses to interpose on those things necessary for a minimal complete checkpoint, comparable to the functionality offered by other user-level checkpointers[36, 40]. These are library-based checkpointers, which require re-linking of the application in order to interpose on its system calls.

Standard I/O. The port references representing the `stdin`, `stdout`, and `stderr` file handles are recognized by the checkpointer during checkpointing and, on restart, are reinitialized with the corresponding file handles in the new environment. Thus, all standard I/O file descriptors (including descriptors in nested subprocesses of the application) are transparently rerouted to the new environment.

Service Ports. When the sub-environment asks for any of the generic service ports—memory allocator, file system, or scheduler—the checkpointer hands back a reference and tracks that reference in an internal catalog. These service ports are handled exactly as the Open Files above.

Special Files. Our current checkpointer doesn't interpose on any file system accesses, but could recognize file open calls and checkpoint file state (or whole files) with the process, in order to provide a more consistent restart.

Unknown References. References to things the checkpointer chose not to intercept, for example arbitrary files, will be replaced with null references. This has similar consequences to an NFS server going down and leaving stale file handles behind.

Process IDs. Since there are no explicit IDs in the RVM model, when restoring a *complete* environment conflicts cannot occur. Process ID troubles can occur when checkpoint/restoring only part of an environment. In particular, since the checkpointer is not currently process-manager-aware, a single process restored under the PM is not assigned a process ID. The process interface and the checkpointer can be easily extended to fix this. Another response to this kind of issue, enabled by the RVM flexibility, is that the user can simply run another copy of the PM under the checkpointer, which then runs the target application.

IPC state. Processes involved in IPC at the time of a checkpoint will restart the IPC when the checkpoint is completed or restored. The act of "stopping" a thread causes the kernel to back the thread's actions out to a re-entry point.

Checkpointer Summary
Two key features of our RVM model facilitate checkpointing. First, the exportable state of kernel objects allows any application to extract and store the state of kernel objects. Second, the consistent interface provided by the model encapsulates the checkpointer's target to the extent that features previously available only in kernel implementations are feasible outside of the kernel.

6.4 The Debugger and Tracer

We implemented a debugger that can be used to debug either ordinary applications or other virtual machine monitors. The debugger creates a virtual machine containing the process or environment being debugged, and its presence is completely transparent to the code running in that virtual machine.

The debugger works by initializing the keeper port reference of the process to be debugged to a port it creates when the child process is spawned. When a thread in the child faults, the kernel sends an exception RPC along with the thread's register state to its keeper port. The debugger handles this message and via read/write calls on its stdin/stderr, communicating via a serial line with a remote host running GDB. The debugger restarts the thread by sending a reply to the kernel that includes the thread's modified register state.

Note that although Mach 3.0 provides a similar ability to interpose on an exception port, Mach allows a task to change its own exception port reference, unlike Fluke. Thus a buggy or uncooperative Mach task could escape the debugger's control. This is a simple example of the inadequacy of existing kernels for implementing recursive virtual machines.

The Tracer
Finally, we implemented a tracer that can be used to trace the message activity of an arbitrary application nested within the tracer. The tracer interposes on the application's parent port and on any port references the child task receives through the parent port. It does this by creating a new port reference and passing that to the child task instead of the original. The tracer transparently forwards messages received from the child to the original port and vice versa.

Besides monitoring RPC activity to aid in debugging, the tracer can also function as a complete but "null" virtual machine monitor, in that it interposes on every interface, but does nothing except pass data on. This can be used to quantify the worse-case communication overhead.

6.5 Distributed Memory Manager

A distributed memory manager cooperates with other DMMs through IPC to create one large transparently distributed environment out of several independent environments. A DMM is very similar to a virtual memory manager (MM), in that it provides a virtual address space paged to some external storage location. However, whereas a VMM pages things primarily to disk or other stable storage, a DMM pages things primarily to other nodes. Note that the exportability of kernel object state should allow a DMM to distribute entire POSIX-like operating environments, not simply memory.

The DMM and MM functions could be combined into one program or could remain separate. If they are separated, then a DMM could be run either on top of a MM, to provide a distributed memory with each node having separate page-out space, or below a MM, with the MM providing a single common paging space for the entire distributed subsystem.

DMMs could be implemented to support different coherency models; however, the kernel architecture is designed to be able to support release-consistent DSM[7] particularly well. Since all of the kernel objects in use by its subtasks, such as mutexes and condition variables, are fully visible to it, the DMM should have the perfect tools.

In addition, the "segment" abstraction of the Common Protocols provides a handle to determine the granularity of synchronization events. In other words, the CP conventions provide information as to how much memory must be synchronized when a given mutex or condition variable is used.

7 Experimental Results

In order to evaluate the performance effects of recursive virtual machines in our system, we used micro benchmarks, some drawn from the `lmbench` suite[38]. These benchmarks are designed to reveal the performance properties of operating systems that directly affect real-world applications. Our primary interest in these tests is to reveal the performance effect of different VMMs in our system on various types of applications; thus, we are mostly concerned with relative slowdown due to VMMs rather than the absolute performance of the system. All tests were performed on a 100MHz Pentium PC with 32MB of RAM.

	Time (μs)
Null system call	2.0
Mutex lock	5.2
Mutex unlock	5.8
Context switch	1.4
Null cross-domain RPC	54.6

Table 1: Absolute performance of microkernel primitives

Test	Description	Fluke	Linux
`bw_mem_cp`	Memory bandwidth	37 MB/s	42 MB/s
`bw_mmap_rd`	File `mmap` read	47 MB/s	74 MB/s
`bw_file_rd`	Cached file reads	24 MB/s	23 MB/s
`lat_sig`	Signal handling cost	259 μs	52 μs

Table 2: Absolute `lmbench` results for Fluke and Linux

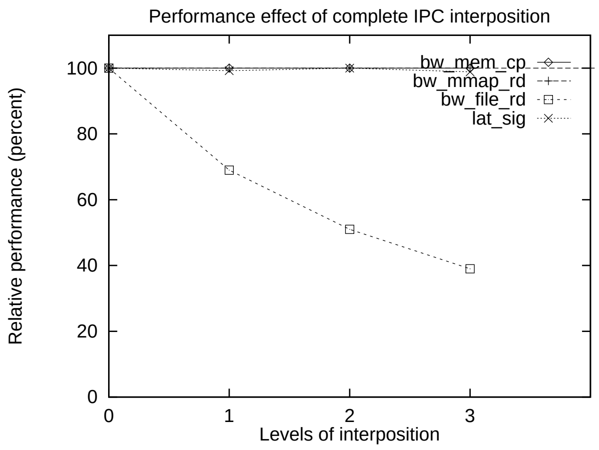

Figure 3: Worst-case slowdown due to IPC interposition

Absolute performance: To provide a baseline for further evaluation, present in Table 1 the absolute times for various primitive Fluke microkernel operations, and Table 2 shows absolute times for the `lmbench` benchmark programs we will use in later tests running directly on top of the microkernel with no intervening VMMs. For reference, we also show `lmbench` performance results for Linux, taken from the original `lmbench` paper[38]. Note that the Linux tests were made on a faster machine (120MHz Pentium) than we used in our tests (100MHz Pentium). Also, since Linux is a mature, well-optimized monolithic kernel while Fluke is a mostly unoptimized microkernel, the performance results for OS-intensive tests are not very comparable. However, since the performance results of primary interest for this paper are the relative numbers presented below, we felt that these discrepancies are not a major issue.

Cost of IPC interposition To measure the worst-case slowdown caused by IPC interposition, we measured the effect on the execution times of various applications caused by the complete IPC interposition done by the tracer; these times are shown in Figure 3. This test also reflects the performance that might be expected from a security monitor VMM that supervises an untrusted application environment. However, this test reflects *worst-case* interposition cost; other types of VMMs such as process managers only need to interpose on some IPC connections, not all. As expected, applications that make heavy use of IPC (e.g., file read) suffer most from this test, while other applications are essentially unaffected.

Performance of Various VMM hierarchies We measured, under a variety of VMM hierarchies, a program which forked a chain of five processes. This utilizes both the process manager and the memory manager ports. We found slowdown to be linear with the number of layers. The marginal cost of the PM is about 24%. The data are consistent with a MM interposition costing a little bit more than a PM interposition.

```
  fw05    diff-fw05     Nesters
 25956940               PM
 52486392 26529452      MM PM
 65405272 12918880      MM PM PM
 82247326 16842054      MM PM MM PM
 97997148 15749822      MM PM MM PM PM
116497487 18500339      MM PM MM PM MM PM
```

8 Conclusion

We have presented a novel approach to providing modular and extensible operating system functionality based on a synthesis of microkernel and virtual machine concepts. We have demonstrated the design and implementation of a microkernel architecture that efficiently supports fine-grained recursive virtual machines. Our prototype implementation of this model indicates that it is practical to modularize operating systems this way: some types of virtual machine layers impose almost no overhead at all, while others impose some overhead, but only on certain classes of applications.

However, it remains to be seen how the model will scale to a real, fully functional system. Although we addressed some of the issues of implementing a nested process model in a Unix-like environment, there are many others, such as networking and security. In order to address these issues, we are working with the Free Software Foundation to port the GNU Hurd, a fully functional Mach-based multiserver OS, to our Fluke kernel, using our RVM model.

References

[1] N. Batlivala, B. Gleeson, J. Hamrick, S. Lurndal, D. Price, and J. Soddy. Experience with SVR4 over Chorus. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 223–241, Apr. 1992.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 1995.

[3] A. C. Bomberger and N. Hardy. The KeyKOS nanokernel architecture. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Seattle, WA, Apr. 1992.

[4] T. C. Bressoud and F. B. Schneider. Hyporvisor-based fault-tolerance. In *Proc. of the Fifteenth ACM Symposium on Operating System Principles*, pages 1–11, December 1995.

[5] M. I. Bushnell. Towards a new strategy of OS design. In *GNU's Bulletin*, Cambridge, MA, Jan. 1994. Also http://www.cs.pdx.edu/˜trent/gnu/hurd-paper.html.

[6] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 165–177, Monterey, CA, Nov. 1994. USENIX Assoc.

[7] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Asilomar, CA, Oct. 1991.

[8] J. Carter, D. Khandekar, and L. Kamb. Distributed shared memory: Where we are and where we should be headed. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, pages 119–122, May 1995.

[9] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 179–193. USENIX Association, Nov. 1994.

[10] G. Deconinck, J. Vounckx, R. Cuyvers, and R. Lauwereins. Survey of checkpointing and rollback techniques. Technical Report O3.1.8 and O3.1.12, ESAT-ACCA Laboratory Katholieke Universiteit Leuven, Belgium, June 1993.

[11] R. P. Draves. A revised ipc interface. In *Proc. of the USENIX Mach Workshop*, pages 101–121, October 1990.

[12] E. N. Elnoxzahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[13] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[14] B. Ford and M. Hibler. Fluke: Flexible μ-kernel environment — application programming interface reference (draft). 121 pp. University of Utah. Available as ftp://mancos.cs.utah.edu/papers/sa-flukeref.ps.gz and http://www.cs.utah.edu/projects/flux/fluke/html/-sa-flukeref/ (HTML format), 1996.

[15] B. A. Ford and S. Susarla. Flexible multi-policy scheduling based on cpu inheritance. Submitted for publication., May 1996.

[16] R. P. Goldberg. Architecture of virtual machines. In *AFIPS Conf. Proc.*, June 1973.

[17] R. P. Goldberg. Survey of virtual machine reseach. *IEEE Computer Magazine*, pages 34–45, June 1974.

[18] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proc. of the Summer 1990 USENIX Conference*, pages 87–96, Anaheim, CA, June 1990.

[19] G. Hamilton, M. L. Powell, and J. J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79, 1993.

[20] N. Hardy. The keykos architecture. *Operating Systems Review*, September 1985.

[21] K. Harty and D. Cheriton. Application-controled physical memory using external page-cache management. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–199, September 1992.

[22] Ibm virtual machine facility /370 planning guide. Technical Report GC20-1801-0, IBM Corporation, 1972.

[23] Ibm virtual machine facility /370: Release 2 planning guide. Technical Report GC20-1814-0, IBM Corporation, 1973.

[24] M. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proc. of the Fourteenth ACM Symposium on Operating System Principles*, pages 80–93, December 1993.

[25] A. V. K. Krueger, D. Loftesness and T. Anderson. Tools for the development of application-specific virtual memory management. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Oct. 1993.

[26] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of hte 1994 Winter USENIX Conference*, pages 115–132, January 1994.

[27] Y. A. Khalidi and M. N. Nelson. Extensible file systems in Spring. In *Proc. of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–14, 1993.

[28] C. Landau. The checkpoint mechanism in keykos. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, September 1992.

[29] H. C. Lauer and D. Wyeth. A recursive virtual machine architecture. In *ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, pages 113–116, March 1973.

[30] C. H. Lee, M. C. Chen, and R. C. Chang. HiPEC: High performance external virtual memory caching. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 153–164, Monterey, CA, Nov. 1994. USENIX Association.

[31] H. M. Levy. *Capability Based Computer Systems*. Digital Press, 1984.

[32] J. Liedtke. Clans and chiefs. In *Proceedings 12. GI/ITG-Fachtagung Architektur von Rechensystemen*, 1992.

[33] J. Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[34] J. Liedtke. A persistent system in real use – experiences of the first 13 years. In *Proc. of the Third International Workshop on Object Orientation in Operating Systems*, pages 2–11, December 1993.

[35] J. Liedtke. On micro-kernel construction. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, Dec. 1995.

[36] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Proc. of the Winter 1992 USENIX Conference*, 1992.

[37] J. H. March. The design and implementation of a virtual machine operating system using a virtual access method. In *AFIPS Conf. Proc.*, June 1973.

[38] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proc. of 1996 USENIX Conference*, page xxx, Jan. 1996.

[39] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria*. 1985. DoD 5200.28-STD.

[40] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proc. of the Winter 1995 USENIX Technical Conference*, January 1995.

[41] G. J. Popek and C. Kline. Verifiable secure operating systems software. In *AFIPS Conf. Proc.*, June 1973.

[42] A. Valencia. An overview of the vsta microkernel. http://www.igcom.net/— —jeske/VSTa/-vsta_intro.html.

[43] D. Wagner, I. Goldberg, and R. Thomas. A secure environment for untrusted helper applications. In *Proc. of the 6th USENIX Unix Security Symposium*, 1996.

[44] C. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusettes Institute of Technology, September 1995.

[45] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. North Holland, NY, 1979.