THE ALTA OPERATING SYSTEM

by

Patrick Alexander Tullmann

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

December 1999

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Patrick Alexander Tullmann

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Frank J. Lepreau

Wilson C. Hsieh

John B. Carter

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Patrick Alexander Tullmann in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Frank J. Lepreau
Chair, Supervisory Committee

Approved for the Major Department

Robert R. Kessler
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Many modern systems, including web servers, database engines, and operating system kernels, are using language-based protection mechanisms to provide the safety and integrity traditionally supplied by hardware. As these language-based systems become used in more demanding situations, they are faced with the same problems that traditional operating systems have solved—namely shared resource management, process separation, and per-process resource accounting. While many incremental changes to language-based, extensible systems have been proposed, this thesis demonstrates that comprehensive solutions used in traditional operating systems are applicable and appropriate.

This thesis describes Alta, an implementation of the Fluke operating system's nested process model in a Java virtual machine. The nested process model is a hierarchical operating system process model designed to provide a consistent approach to user-level, per-process resource accounting and control. This model accounts for CPU usage, memory, and other resources through a combination of system primitives and a flexible, capability-based mechanism.

Alta supports nested processes and interprocess communication. Java applications running on Alta can create child processes and regulate the resources—the environment—of those processes. Alta demonstrates that the Java environment is sufficient for hosting traditional operating system abstractions. Alta extends the nested process model to encompass Java-specific resources such as class files, modifies the model to leverage Java's type safety, and extends the Java type system to support safe fine-grained sharing between different applications. Existing Java applications work without modification on Alta.

Alta is compared in terms of structure, implementation and performance to Fluke and traditional hardware-based operating systems. A small set of test appli-

cations demonstrate flexible, application-level control over memory usage and file access.

Dedicated to my parents.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I thank my thesis committee for assisting in the design and description of Alta; implementing Fluke in Java was conceived by my committee chair, Jay Lepreau. Mike Hibler and Bryan Ford deserve special credit for designing and implementing Fluke, in addition to their suggestions and ideas for Alta. Godmar Back, Eric Eide, John McCorquodale, Sean McDirmid, and Alastair Reid all helped me with Java and helped me refine my design for Alta.

Tim Wilkinson and Glynn Clements, whom I have never met, deserve recognition for building the systems—Kaffe and Kore, respectively—that I used as foundations for Alta.

Finally, I must thank Colette Mullenhoff for her support and understanding for the duration of this thesis.

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Department of the Army under contract DABT63–94–C–0058, and the Air Force Research Laboratory, Rome Research Site, USAF, under agreement F30602–96–2–0269.

CHAPTER 1

INTRODUCTION

In operating systems, the *process* is the abstraction that separates the kernel from applications and applications from each other. The process is the scope of resource management and identity; for example, most operating systems impose per-process memory limits and perform per-process access checks. Traditionally, process separation—the principle of preventing malicious or incorrect code from accessing protected data—has been enforced by hardware protection mechanisms. The memory mapping hardware in these systems can check individual read or write accesses to a page of memory. By manipulating the memory mapping permissions, a kernel can give different access privileges to different processes. Such low-level enforcement successfully isolates both malicious and faulty processes. Fine-grained sharing, however, is awkward, expensive, or impossible because data sharing between processes can only be protected on coarse, page-based boundaries. This property of hardware-supported operating systems, coupled with the cost of switching permission tables, encourages an application structure that minimizes the number of process "boundary crossings" and reduces interprocess sharing to coarse page-based methods [49].

As an alternative to hardware-enforced separation, software-based separation methods perform memory access permission checks in software. Cedar [44] from Xerox PARC was the first to rely on software-based protection; Cedar allowed users to change or add system features via "extensions." Extensions in Cedar and other systems are externally defined modules that extend or modify the behavior or available functionality of a system. Other extensible systems include SPIN [9], the HotJava WWW browser [42], Oberon [52] and even the Emacs editor [41]. While web browsers and text editors are not operating systems in a traditional sense, both of these systems may need to manage and control their extensions in the same manner as an operating system.

Software-based protection mechanisms are also useful in systems that have no memory protection hardware. For example, the Apple Newton's NewtonOS [50] uses software-based protection mechanisms to separate the "kernel" from applications because the Newton does not include hardware for enforcing memory protections.

Software-based protection mechanisms can take several forms, including type-safe languages, annotated code systems, and checkable (or provable) code accompanied by a proof. All of these systems use compile-time, load-time or run-time checks to prevent a program from making illegal memory accesses. Annotated code systems, such as SFI [2], and provable code systems, such as PCC [34], have the advantage that they support legacy programming languages as the annotations (or proof) can be generated after the code is compiled. Currently, safe languages are more mature than either annotated code or provable code systems.

Although most software-based operating systems are defined in terms of "extensions" and operating system models are defined in terms of "processes," the only distinction seems to be a subjective measure of the size or independence of the module. A running application is usually considered a process while a modification to a larger system is usually considered an extension, but the distinction is arbitrary and the terms can be interchanged.

Existing software-based operating systems tend to be designed for single-user environments. Cedar, for example, was designed to make the operating system of the Xerox Alto computer extensible by the sole user of the system, thus avoiding many trust and safety issues. Oberon [53] makes a critical design assumption that restricts it to a single-user environment: tasks are not preemptible. The protections provided by languages for these inherently single-user environments can effectively shield the system from the majority of erroneous applications, but provide opportunities for malicious programs to bring a system down. As operating systems employing software-based protection mechanisms become used in more widely networked and multiuser environments, protecting a system from merely incorrect applications is insufficient. Protection needs to be extended to contain malicious and destructive applications, especially in the area of resource management.

Protection from and separation of mutually untrusting applications is the same problem that traditional, hardware-based multiuser operating systems face, and thus solutions from that domain should be applicable to language-based operating systems. To demonstrate that language-based extensible systems can benefit from the models developed for traditional operating systems, this thesis incorporates the process model and API from the Fluke operating system [20, 22] into Alta, a new virtual machine based on the Java virtual machine (JVM) [27]. The result is a JVM that supports nested processes and direct, hierarchical resource controls—the same set of services and protections that the Fluke model provides in a hardware-supported microkernel.

I chose to base Alta on the Fluke process model, or nested process model, for several reasons. First, the model incorporates per-process resource controls. Second, the nested process model infrastructure enables system services to be provided outside of the kernel, a feature appropriate for mobile-code platforms [45]. Third, the model is a complete process model: it specifies interprocess communication and synchronization primitives in addition to the basic process abstraction. Finally, I have experience with Fluke [21, 22, 46, 47].

Java was chosen as the "host" language for Alta because it provides the type-safety, language-level access control, and run-time environment necessary for an extensible system [32]. Java is also popular, and is used in a number of widespread systems, such as web servers and web browsers, that can benefit from more comprehensive support for extensions and resource controls.

The term "virtual machine" has different meanings in the context of Fluke, Java, and Alta, all of which are fundamentally similar. In Java the phrase "virtual machine" usually describes the Java bytecode interpreter/compiler and run-time support because these components act as a machine simulator. In Fluke, which is modeled on recursive virtual machines [26], the phrase "virtual machine" is defined as the environment that encapsulates a process because the process runs in the context of a virtualized machine. Both of these definitions are, in essence, about the presentation of a machine-like, virtual interface for an application. In this thesis, the phrase "virtual machine" describes a generic Java virtual machine.

1.1 An Alta sandbox

The remainder of this chapter describes the Alta sandbox, an extended example of how an application might take advantage of Alta's process model. A *sandbox* is a controlled environment for untrusted code to execute in. The code gets to play the in safe and simple sandbox but is not allowed to directly access many, potentially unsafe, operations.

The Java applet sandbox, is the basis for a web browser's ability to control arbitrary, downloaded code. The sandbox is designed to contain applets and restrict an applet's access to critical system resources. The original Java sandbox, as implemented in JDK 1.0.2,[1] provides a very strict access policy: applets are not allowed to access the local file system in any way, and they may only use network facilities to connect back to the host from which they originated. In later implementations of the JDK, an appropriately identified applet can be placed in a less restricted sandbox or even completely escape the sandbox.

Redefining the applet framework to take advantage of Alta's nested process features demonstrates two aspects of Alta. First, the Alta sandbox demonstrates that Alta provides sufficient mechanism to satisfy the containment requirements of the Java specification; second, the sandbox demonstrates application-level control over other applications. Additionally, the Alta sandbox provides a number of new features beyond the original. Applications running in the sandbox are only able to use the amount of memory granted to them. Each sandboxed application has its own separate static variable namespace. Sandboxed applications can contain

[1] The JDK is Sun Microsystem's Java Development Kit which is the reference implementation for any given Java version.

sub-sandboxed child applications, and can use the `java.lang.ClassLoader` (for dynamically loading classes) and can create `java.lang.ThreadGroup`s. These last two abilities are unavailable in the basic Java sandbox.

The Alta sandbox uses three mechanisms to contain and control processes. First, per-process static member data, per-process memory limits, and per-process class namespace control are directly supported by Alta. Second, simple limits are enabled by controlling the class namespace of the sandboxed process. For example, to completely deny access to the local file system, a sandbox could map the file access classes in its child to classes that throw an exception if any methods are invoked. The third mechanism for containing child processes is the Alta capability system. For example, to enforce a policy that denies network access after an application has accessed the local file system, a sandbox could use the class namespace controls to install replacement classes for the file and network access classes. These replacements would use capabilities and IPC, in place of native method calls, to perform the file or network operation. Thus a sandbox could make dynamic decisions about granting or denying access to the network based on the file access patterns of the application. (Section 3.2.2.1 will explain this process in more detail.)

1.2 Roadmap

Chapter 2 details Fluke's nested process model and provides a short introduction to Java. The design of Alta is outlined in Chapter 3, followed by results and analysis in Chapter 4. Some future directions for Alta are listed in Chapter 5. Chapter 6 discusses other work that supports a process abstraction in Java and other language-based systems, along with work in safe languages and extensible systems in general. Chapter 7 presents conclusions that can be drawn from this work. Appendix A documents the Alta kernel API and Appendix B documents the interposable IPC-based API (IPCIF API).

CHAPTER 2

BACKGROUND

This chapter provides a brief introduction to Java and provides background on the design and development of Fluke's nested process model.

2.1 Java

Java is an object-oriented, type-safe, garbage-collected language that directly supports synchronization, multithreading, and exceptions [32]. Java source code is generally compiled to bytecodes (pseudo-instructions for a virtual machine) that are verified and executed by a Java Virtual Machine [27]. Because of the type-safety provided by the language, compiler, and verifier, Java is a reasonable environment in which to run untrusted code. Just as the use of memory protection hardware by a standard operating system provides protection from a large class of malicious or buggy code, the Java language and the verifiers for compiled bytecodes provide a similar level of protection.

While Java is designed to provide enough safety to run untrusted programs, available implementations and parts of the design have well-known flaws [16, 40]. Other work is being done on improving bytecode verification [28, 39] and on other aspects of Java security [49]. For the purposes of this thesis, I assume that the verifier and interpreter do their jobs correctly. The implementation of Alta is based on a freely available JVM, Kaffe [51], which currently has neither a working verifier nor support for some of the language-level access controls.

Java applications run in an environment that has two basic components: the Java virtual machine [32] and the standard Java libraries [11, 12]. The virtual machine interprets or compiles bytecodes and provides threading and garbage collection. The standard Java libraries implement the various Java programming interfaces and standard services. Many of the classes in the standard Java libraries are implemented entirely in Java and are independent of the virtual machine. On the other hand, many of the core Java classes—those in the `java.lang` package—are tightly integrated with the implementation of the virtual machine. To access underlying system services, such as the file system or the network, classes must utilize native methods—methods that are not implemented in Java.

Because "Java" is both the name of the Virtual Machine interpreter/compiler and the language I will refer to the interpreter as the Java virtual machine or "JVM." When I use "Java" unadorned, I mean the language.

2.2 The Fluke nested process model

The Fluke nested process model is an architecture for organizing and controlling processes that compete for resources. Processes request services and resources from "ancestor" processes, which can grant, deny, or pass on requests. The model provides explicit, low-level support for the two most critical resources: memory and the CPU. More abstract resources, such as files, network access and processes, are controlled through a set of capability-based interfaces: capabilities are the basic mechanism for communication between processes. Figure 2.1 diagrams the nested process model.

The nested process model is implemented in Fluke [22], a prototype microkernel written in C that supports POSIX-like processes that access high-level services through an IPC-based API; primitives and basic services such as synchronization and scheduling are provided directly by the kernel. Processes are separated by standard hardware memory protection—each process runs in its own address space. Because cross address-space communication is heavyweight, applications must provide a great deal of functionality for each cross-process invocation; thus, Fluke is a suitable environment for virtual memory management, process management, checkpointing, file systems, and other heavyweight services. What Fluke is not suitable for is small, communicative components that share complex data structures.

Processes in the nested process model can be "nested": a completely encap-

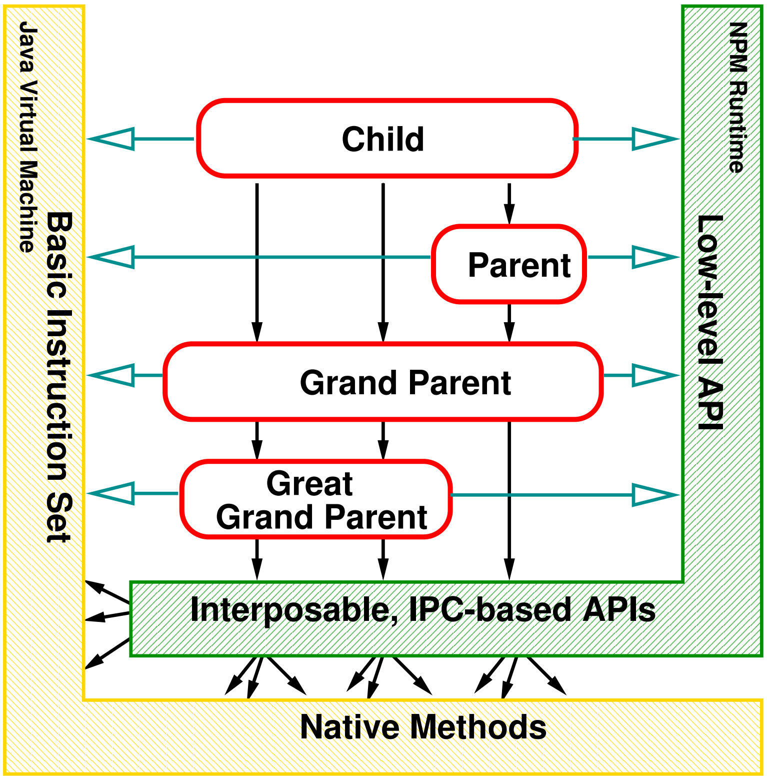

Figure 2.1. A nonartist's visualization of the nested process model in Alta. The two L-shaped boxes represent the trusted base of the system: the virtual machine (the left-side and bottom-most L) and the "kernel" or nested process model runtime (the right-side ⅃). These two components contain the processes (the oval-shaped boxes); older processes (ancestors) are closer to the bottom of the box. The vertical arrows from each process to its ancestors represent access through IPC-based interfaces. In this example, the youngest process is getting two services from its grandparent and one service from its immediate parent. At all levels of the process hierarchy, a process has direct access to the basic virtual machine instructions and the low-level kernel API.

sulated process requests services and resources exclusively from its parent. Total encapsulation benefits the parent process, as it is assured of what resources the child process uses. Additionally, encapsulation enables the child process to manage its own children, while the parent is oblivious to the presence of anything more complex than a single child running within it. Nesting enables applications to contain whole hierarchies of processes, and for different applications to be composed together in a system.

While the focus of the nested process model is hierarchical nesting and the notion of a hierarchy of processes is explicitly supported by the model, it is worth noting that interprocess communication need not follow the hierarchy. For example, in a system where a common ancestor process is willing to receive capabilities from child processes and hand them out to arbitrary children, then any processes in the system that can communicate with the ancestor process can get capabilities to communicate directly with each other. Figure 2.2 illustrates a specific instance of this communication. Additionally, while resource allocations are generally hierarchical, it is not necessary to explicitly manage each resource at each level of the hierarchy. For example, a parent process can spawn two child processes and interpose only on their file system accesses and not on any other resource; the children only pay a cost for the interposed accesses. Continuing with Figure 2.1, the "Parent" process is interposing on only one interface and letting "Child" communicate directly with

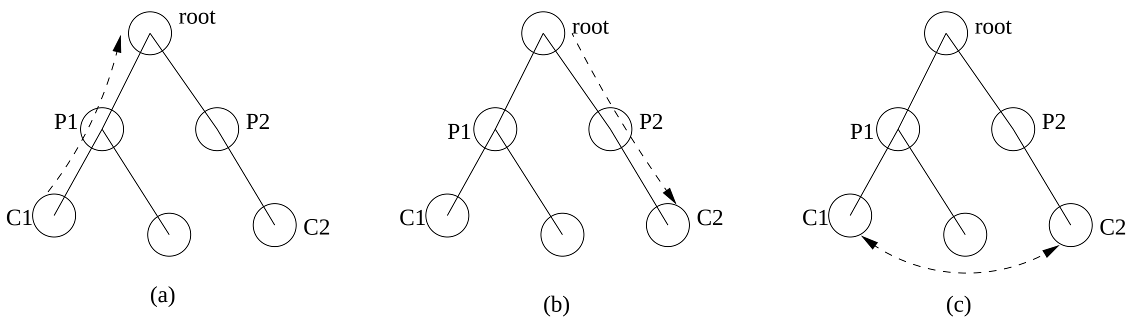

Figure 2.2. Establishing a communication path independent of the process hierarchy. Process C1 can set up direct communication with process C2 in the following way. (a) Process C1 sends a capability to the root process (or any other shared ancestor of C1 and C2). (b) Process C2 obtains the capability from the root process. (c) C1 and C2 can communicate directly with each other. The straight lines represent the ancestor relationship between a process and its parent. The dashed arcs represent the path that IPC operations follow. Note that this is feasible only if processes P1 and the root cooperate for step (a), and processes P2 and the root cooperate for step (b). Any of these processes can refuse to share the capability, thereby stopping C1 and C2 from communicating directly.

"Grand Parent" on the other two interfaces.

To support user-level virtual memory servers and flexible, user-level checkpointers [46], Fluke provides a mechanism for a process to extract the kernel "state" of any low-level object (see Section 3.2.7). It is not necessary that all actual kernel state be exported, but that any state not exported can be derived from state that is exported. For example, in Fluke a parent must be able to get enough state from a thread to create a new thread that is indistinguishable from the original. To support such state transparency, the kernel, in addition to providing all the state, must provide the state in a clean and timely fashion. For example, the kernel cannot suspend a parent process indefinitely while the parent accesses some state, as this would give a child some measure of control over its parent.

There are three distinct components of the nested process model programming interface. First is the basic instruction set in the underlying "machine"—hardware machine instructions (in Fluke) or virtual machine instructions (in Alta). The second component is the low-level programming interface that provides basic abstractions for managing memory and threads, plus the basic infrastructure for communication, sharing, and synchronization between processes. The third component is the set of high-level, interposable interfaces for standard OS abstractions. The first two components are always directly available to all applications, while the set of interposable interfaces available to an application is controlled by its ancestors.

The basic instruction set is usually predefined and cannot be modified to support the nested process model. The instruction set should not, however, contain instructions that operate on global state, as those operations will not be subject to interposition. For example, the `mov` instruction is allowable because it only affects the provided operands. On the other hand, instructions such as the Intel x86 `gettsc` instruction are not nestable because they return global information—in this case, a global notion of time.[1] Generally, privileged instructions (such as I/O instructions) need to be made subject to interposition and should only be directly

[1] There are certainly solutions for problematic instructions such as `gettsc`; in this case it may be desirable to simply leave it as an accessible instruction.

executed by the kernel.

The low-level API that provides the basic abstractions in the system has the same restrictions as the basic instruction set. Thus, all operations at this level operate only on provided state—that is, the operations manipulate state provided by the caller. (These operations may impact "global" kernel state, but only it is soft state and never directly visible outside the kernel.) Table 2.1 shows each object supported by the low-level Fluke API. Generally, each object supports create, destroy, and reference operations in addition to object-specific operations. Note that no object provides direct access to any resources. For example, the memory mapping primitives require that another process provide the memory to be mapped. (In Fluke, a special kernel process bootstraps this recursion.)

The third component, the IPC-based protocols, is where the nested process model resembles traditional capability-based systems and is where processes are able to exert control over other processes through interposition. Well-defined protocols should be amenable to flexible interposition (e.g., a process should not have to interpose on the memory interface if it only wants to interpose on file system access) and should match the style of high-level API used by system processes. In Fluke, for example, the IPC-based protocols are designed to support a POSIX-compatible standard C library. Appendix B provides more details on Alta's

Table 2.1. The kernel objects defined by Fluke.

Object	**Description**
`Space`	Container for threads and memory.
`Thread`	A thread of control in a `Space`.
`Region`	A contiguous range of virtual memory that can be exported.
`Mapping`	A contiguous range of virtual memory that is imported.
`Port`	A destination for client thread IPC requests.
`Port Set`	A collection of ports on which a single (server) thread can wait.
`Reference`	A cross-process reference to another object (a capability).
`Mutex`	A safe, cross-process mutex.
`Cond`	A safe, cross-process condition variable.

IPC-based protocols which were defined to match the requirements of the Java standard libraries.

CHAPTER 3

DESIGN OF ALTA

Alta implements the nested process model in a Java Virtual Machine. To accomplish this merger both the process model and the virtual machine's existing execution model were modified.

To implement a process abstraction in Java, Alta introduces new core objects to Java. Existing Java objects that provide rudimentary support for separation and control, such as the `ClassLoader` or `ThreadGroup`, are insufficient. Section 3.2.2.1 will explain how `ClassLoader`s do not provide the level of control over the name to class binding that is required by Alta. `ThreadGroups` in Java are merely a convenient mechanism for stopping a related group of threads. Adding the memory control, IPC and disjoint typespaces required by Alta on top of existing Java classes would prevent Alta from being backward-compatible with existing Java applications. For example, making the `ThreadGroup` the process container would fundamentally alter the meaning of creating a new `ThreadGroup`.

This chapter is divided into two parts: an overview, in six parts, of the design of Alta, followed by a detailed analysis of those six features.

3.1 Overview of the nested process model in Alta

This section outlines six features of Alta, compares them with similar features found in traditional JVMs and Fluke, and then sketches a short motivation for each. Figure 3.1 presents a high-level view of the Alta architecture. The base of the figure shows the host system for the Alta virtual machine: a traditional operating system or a library layer such as the Flux OSKit [19]. Based on this foundation is the Alta virtual machine—the virtual machine and the core native methods. Above this are the Alta object libraries and the standard Java libraries. The IPC-based interfaces,

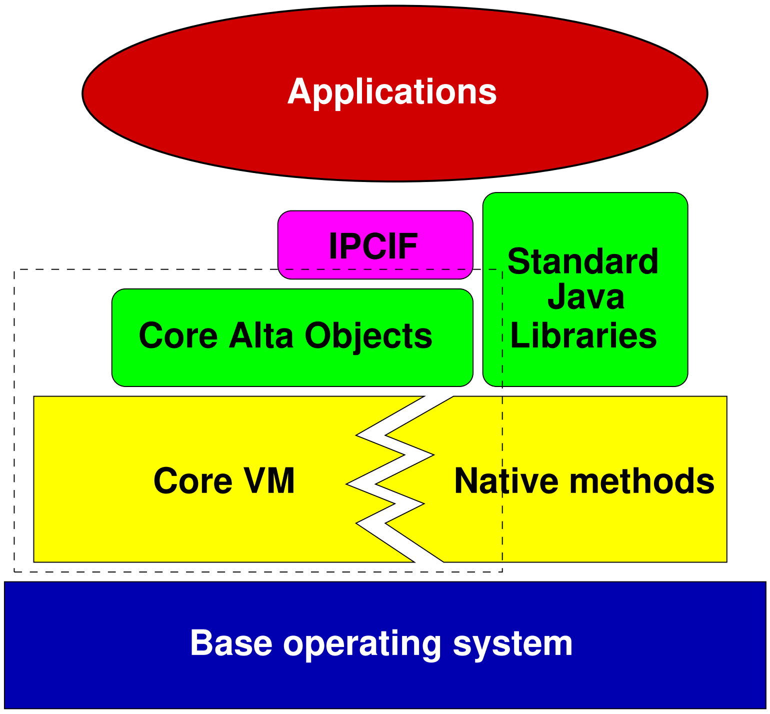

Figure 3.1. A high-level view of the module dependencies and relationships in Alta. At the top, the applications depend on every component beneath them. The IPC-based library (IPCIF) provides services using the core Alta library and the standard Java libraries. These libraries (written in Java) are all dependent upon the virtual machine and the basic native methods. Beneath the virtual machine is the traditional host operating system and its standard libraries. The dashed box represents the Alta kernel.

or IPCIF layer, is the last layer before the applications (in Fluke these interfaces are know as the "Common Protocols").

The Alta kernel is represented by the components inside the dashed box on the left side of the figure. Crossing this boundary is effectively equivalent to a kernel system call trap in a traditional operating system [5].

3.1.1 Each process is a complete JVM

The first goal for Alta is to provide the illusion of a complete Java virtual machine to each process in the system—just as any operating system strives to convince each process that it owns the whole machine. In Java, preserving this illusion requires that each process have its own set of static variables, its own set of classes, and its own set of basic objects (such as `System.out`, a root thread group, etc.).

3.1.2 Alta typespaces

To support the nested process model, the Alta class loading mechanism gives a parent process complete control over the binding of fully qualified class names (such as "java.io.File") to class files. The mapping of class names to class files in a process defines the *typespace* of that process. In Alta, each process has its own private typespace. The parent process is responsible for resolving requests to bind a class name in a child's typespace.

3.1.3 Interprocess sharing

Even though different processes may have disjoint typespaces, the virtual machine has sufficient information to allow safe sharing of objects between processes. Allowing processes to share fine-grained objects efficiently and safely is potentially critical to the performance of a safe, language-based operating system [49]. Interprocess sharing is naturally in conflict with process separation and complicates resource control. The hierarchy of the nested process model provides a simple mechanism for processes in Alta to control the degree of separation (and conversely the degree of sharing) in the system.

3.1.4 Core API implementation

The low-level nested process model API is presented to Java applications as the set of classes in the package `edu.utah.npm.core`, defined in Appendix A. Implementation of the code supporting this API—the kernel code—required similar techniques for protecting it from user processes as are required in a traditional kernel. Additionally, this API is based on the original API designed for Fluke, with modifications to account for Java in the following three areas. First, processes may send Java object references through IPC; second, memory management is accomplished at a higher level of abstraction in Alta; third, Alta does not support state extraction or injection for any kernel objects.

3.1.5 Core resource controls

The major design goal for Alta is to provide a framework for comprehensive resource accounting and control to Java processes. The two core resources explicitly managed by the kernel in the nested process model are memory and CPU. Alta was designed to account for as much of the memory used by a process as possible. For example, the memory used to retain the JIT version of a method is stored in memory charged to the process that performed the JIT compilation. The nested process model specifies a flexible CPU management scheme, CPU inheritance scheduling [23], which I have not implemented in Alta. CPU inheritance scheduling appears to be a good match for Alta but the implementation effort to support it is beyond the scope of this thesis.

3.1.6 IPC-based interfaces

The Alta IPC-based interfaces are based around the native methods used by the core Java libraries. That is, the IPC-based interfaces have been tailored to the requirements of the core Java libraries. Native methods have been grouped and interposition is accomplished on a group basis. For example, there is a group of operations for the file system, a group for individual file I/O, a group for memory management, etc. The grouping of the IPC-based interfaces in Alta is comparable to the grouping of the POSIX oriented, IPC-based interfaces in Fluke.

3.1.7 Exportable state

Fluke supports the full extraction of kernel object state, which is critical for complete virtual memory managers and multi-process checkpointers [46]. Alta does not support exportable state of kernel objects. The type-safe extraction of a thread stack is well beyond the scope of this thesis (and perhaps impossible). Support for fully exportable state is required for a number of user-level services but, as Alta demonstrates, it is not a critical component of the nested process model.

3.2 The nested process model in Alta

This section provides a detailed review of each of the six areas outlined in the previous section. Features that are derived from existing Java virtual machines are compared to those found in JVMs; features that are derived from traditional operating systems are compared to those found in Fluke and other traditional systems.

3.2.1 Each process is a complete Java virtual machine

Alta processes are designed to run whole Java applications, unmodified. All of the services and capabilities available in a traditional Java virtual machine are available to each process in Alta. Each process has its own root `java.lang.ThreadGroup`, its own root `java.lang.ClassLoader`, and any process may define its own classloaders. All threads in the system are associated with exactly one process, memory is divided on a per-process basis, and each process defines a typespace. Finally, each process owns a port reference, called the keeper port, where the process can make initial requests for services, resolve names in its typespace; and signal when all its threads have died.

In Alta, each process has its own copy of the static data associated with a class, maintaining the illusion that each process has a separate virtual machine. Static data is associated with a class but not with any instance of that class. In existing Java virtual machines, static data is effectively global data (though visibility of the data may be controlled by language-level access modifiers, such as `private`).

As an unfortunate side-effect of the per-process static data and the separate typespaces, each process in Alta maintains its own copy of JIT (Just In Time) compiled methods. (Typespaces are explained in the next section.) The JIT compiler in Kaffe—the virtual machine Alta is based on—inlines static variable references into generated code. Thus a reference to the static variable `java.lang.System.out` in a method would be JIT-compiled into a direct reference to the address of the `java.lang.System.out` static variable. Additionally, separate typespaces mean that class hierarchies may differ between typespaces, and the compiled methods must reflect this difference. Per-process JIT compiled methods are effectively equivalent to using statically linked binaries in traditional operating systems. Changes to the JIT compiler in Alta to make static variable accesses indirect and to share code between equivalent typespaces are areas for future work.

Separation of processes is the cornerstone of resource control in Alta. Resources are allocated on a per-process basis in Alta, and the distinction between processes is critical for accounting resource consumption and release.

Given that processes are separated, the system must provide a facility for interprocess communication (IPC). Alta uses the IPC semantics specified by Fluke. Thus, while each process is presented with the illusion of running in its own Java virtual machine, facilities for discovering and communicating with other processes in the machine are available.

3.2.2 Alta typespaces

Alta extends the nested process model to encompass Java's classes. Classes are Java objects that define the object layout and object methods available in a typespace. Control over the typespace of a process provides the ultimate control over that process and enables parent processes to exercise precise control over resource accesses and to effectively "download" resource management into a child process.

A class is defined by a set of class files. The *class file format*, defined in the Java Virtual Machine specification [32], specifies the external representation of a class.[1] This format allows classes to be stored and transferred between Java virtual machines. A class file symbolically specifies a superclass, fields, and methods. Class files can be dynamically loaded into the system, which extends the set of classes at run-time. When a class file is loaded, the virtual machine resolves the symbolic references to other class files, performs a series of link-time checks to make sure that the resulting class obeys safety constraints, and creates a `java.lang.Class` object representing the new class [32]. Resolution of a symbolic reference can cause

[1] Technically, the class file format specifies either a Java class or a Java interface. I will refer to them both as "class files."

additional class files to be loaded. The JVM specification defines which names are resolved and when during the loading, linking and execution phases of a class. Note that a class file is not required to be an actual file, it is just a sequence of bytes.

3.2.2.1 Class name resolution in Alta

In Java, the default mechanism for resolving a symbolic name into the appropriate class file is to convert the symbolic name to a file-system-friendly format and look for it on the local file system. For example the name `java.util.Vector` is transformed to `java/util/Vector.class`, which should be found on the local file system. The JVM specification describes an extension mechanism for Java applications to resolve names by subclassing `java.lang.ClassLoader`. Resolution of a name can be delegated to another class loader (usually the system class loader) [31]. Once resolution of a name is delegated, all names referenced from that class file are implicitly delegated, also. This implicit delegation preserves the consistency of the typespace. For example, given the partial definition of the class `C` in Figure 3.2, once a classloader delegates resolution of the name "C," the classloader passes up the opportunity to resolve any name referenced by the class file. In this example, by delegating resolution of the name "C", the name "Related" will also be implicitly delegated.

Alta adds a new level of context—the process—to the type system. Within a process the type rules are unchanged from a traditional Java virtual machine. The type rules between processes, needed for interprocess sharing, are discussed in

```
class C
{
        public Related r;
}
```

Figure 3.2. A simple class `C` that contains a reference to the class `Related`. In a traditional JVM, delegation of the resolution of the name `C` would implicitly delegate resolution of the name `Related`. In Alta this is not so.

Section 3.2.3.

In Alta, when a process attempts to resolve an unresolved symbolic name, it performs an IPC to its parent process and the parent can reply with any class file to which it has access. Regardless of where the parent acquired the class file, that parent will also be responsible for resolving every name referenced by the class file. For example, in response to a request for the class file to bind to the name "java.io.FileInputStream" a parent could respond with the class bound to the name "java.io.FileInputStream" in its typespace or, perhaps, with the class bound to "edu.utah.npm.stubs.NoFileAccess." (For convenience, the parent may reply with a class object, the virtual machine then uses the appropriate class file information in the child.) In either case, all unbound names referenced by the given class file will also have to be resolved. With this mechanism, a parent process has complete control over the nonsystem classes in a child typespace. Implicitly, supporting this level of control requires that the name associated with a class file be per-process and *independent* of the class file. Substituting class files under different names requires no new link-time safety checks as current virtual machines must already perform all of the necessary link-time safety checks to prevent version skew problems. Modifications to the run-time type checks are discussed in Section 3.2.3.

To preserve the integrity of the virtual machine, critical system classes are loaded into each process and bound to their "correct" names by the virtual machine before any process begins execution. Preloaded classes are fixed in the class namespace of a process. There are two kinds of critical classes in Alta. The first set of critical classes is those that define the "kernel" of Alta—all the classes in the `edu.utah.-npm.core.*` package, defined in Appendix A. The second set of preloaded classes is the relatively small number of classes that are entwined with the operations of the virtual machine. Table 3.1 lists those classes. Critical classes must be preloaded to prevent a parent and child process from colluding to break the kernel. Additionally, to protect package-private interfaces in the kernel, the virtual machine must prevent other classes from being loaded into the `edu.utah.npm.core` package.

Table 3.1. The set of critical classes preloaded into each Alta process.

Class	**Justification**
Primitives	All of the primitive types and arrays of primitive types are predefined in each process.
`java.lang.Object`	The common superclass of all objects in a JVM.
`java.lang.String`	`String` is used in a number of critical places. For example, a `String` object represents the class name in class name resolution requests.
`java.lang.Class`	`Class` is used in the process of resolving class names through IPC, so it cannot be dynamically loaded.
`java.lang.System`	`System` is used during process initialization.
`java.lang.Throwable`	The JVM understands the layout `Throwable` as it must fill in and print stack backtraces.
`java.lang.Error`	The JVM generates `Error` objects internally.
`java.lang.Thread`	The JVM understands the layout of Thread objects.
`java.lang.Runnable`	`Thread` requires `Runnable`.
`java.lang.ThreadGroup`	The JVM creates the `ThreadGroup`s.
`java.lang.Cloneable`	`Cloneable` is used by the native clone operation.
`java.util.Vector`	Required by `ThreadGroup`
`java.util.Hashtable`	Required by `String`
`java.util.Dictionary`	Required by `Hashtable`
`java.util.HashtableBucket`	Required by `Hashtable`
`java.io.NI_FileDescriptor`	`NI_FileDescriptor` encapsulates an actual file descriptor and is therefore a trusted class.
`edu.utah.npm.core.*`	See Appendix A

3.2.2.2 Native methods

One complication related to allowing a parent process to control class loading is controlling access to native methods. Native methods are declared in a Java class file but are implemented in C. Access to native methods must be controlled because their code is not under the control of a parent process—native methods can potentially violate the nesting hierarchy. Many native methods, however, pose no problem. For example, the native array copy method, which copies the contents of one array into another, provides no special privileges to the caller. In contrast, a native method such as `java.lang.System.exit()`, which causes the virtual machine to terminate, implies a great deal of privilege.

As an example, the critical class `java.lang.Class` contains the static native method `findSystemClass`, which takes a `String` as a parameter and returns a `Class` object. This method is not implemented in Java, but in C; the virtual machine invokes the C function `java_lang_Class_findSystemClass()` when Java code invokes `java.lang.Class.findSystemClass()`. The simple name transformation that converts a Java native method name into a C function name implies that Alta applications only need to generate a class with an appropriate name to call into native code. Fundamentally, the problem of controlling access to native methods stems from the assumption in the virtual machine that the binding from a name to a class is static. The C code is written for a particular class with a particular name.

Native methods are a useful point for a parent process to interpose on a child. For example, interposing on the exit method and replacing it with a method that uses IPC allows a parent to transform the JVM termination function to a process termination function. In Alta, native methods are semantically grouped and defined separately from the classes that use them. This grouping simplifies interposition for a parent process because all of the relevant methods, and only the relevant methods, are grouped together in a single class. Since these methods are all native methods they are denoted with an "NI" in their name (for "Native Interface"). Specifically, when a parent wants to interpose on the exit method, the native interface `java.lang.NI_Exit.exit()` is interposed on. In Alta, `java.lang.-NI_Exit` is a simple class that contains only an exit method. The default `java.-lang.System.exit()` implementation calls out to `java.lang.NI_Exit.exit()`. If `java.lang.System.exit()` did not invoke `java.lang.NI_Exit.exit()`, but instead was a native method itself, then to interpose on this function would require a parent process to re-implement the whole `java.lang.System` class.

The complication introduced by native methods is best illustrated by an example. Consider a parent process `ProcA`. In order to contain its child processes, `ProcA` replaces the `java.lang.NI_Exit` class with a class that terminates the calling process and not the entire virtual machine; however, if a child of `ProcA` creates its own child, `ProcC`, and maps `java.lang.NI_Exit` in `ProcC` to a custom class containing a native method `exit()`, then when `ProcC` executes `java.lang.-NI_Exit.exit()` it will invoke the native method that shuts down the whole virtual machine, circumventing the controls placed by its ancestor, `ProcA`. This exploit is only possible if the virtual machine does not make a distinction between the class bound to `NI_Exit` in `ProcC` and the class bound to `NI_Exit` in `ProcA`. In Alta, the critical distinction the virtual machine makes is based on the origin of the class files. The class file in `ProcA` was (assuming it should have access to the native method) loaded by the kernel from the local file system, whereas the class file in `ProcC` was created in the standard Java fashion for creating dynamic classes—calling `Classloader.defineClass()`. The assumption is that classes loaded by the kernel are more trusted than arbitrary classes, and a parent process can control a child's access to classes loaded by the kernel. So, in this example, `ProcC` would be unable to execute the native method associated with `java.lang.NI_Exit.exit()`, because `ProcA` denied its child processes access to the trusted class that does have access to the native method. Alta effectively binds native methods to only the trusted class that the native method was written to work with; no other class, whatever its name, may call that native method.

Table 3.2 lists all of the core Java classes containing native methods in Alta. All other classes in the `java.lang`, `java.io`, and `java.util` packages are implemented

Table 3.2. The set of native interface classes defined for Alta's core Java libraries.

Native Interface	**Description**
`java.lang`:	
`NI_ClassLoader`	Contains the native methods used by a `java.lang.-Classloader`: `defineClass()`, `resolveClass()`, and `findSystemClass()`. The first two are safe while `findSystemClass()` is intercepted by the kernel and transformed into an IPC.
`NI_Exit`	The `exit()` method, which is invoked only by `java.-lang.System.exit()`.
`NI_GC`	Functions to invoke the garbage collector, invoke the finalizer, and query for the amount of memory.
`NI_Library`	Functions for dynamically linking native libraries.
`NI_Process`	Functions for manipulating traditional, external processes; only used by `java.lang.Runtime.exec()`.
`NI_SecurityManager`	A function for querying the current thread's stack. Only used by the security manager.
`NI_SystemProperties`	Functions for retrieving basic properties of the system; only used to initialize `java.lang.System`.
`NI_Time`	The function `currentTimeMillis()`.
`java.io`:	
`NI_FileDescriptor`	The basic object used to represent a file in Java. A critical class preloaded into every process.
`NI_FileIO`	Functions that perform basic file I/O operations on a `NI_FileDescriptor` (e.g., `read` or `write`).
`NI_FileSystem`	Functions that perform file system operations. For example, `open()` or `isDirectory()`.
`java.net`:	
`NI_Host`	Functions that provide information about the local host.
`NI_NetIO`	Functions providing network operations on a `java.-io.NI_FileDescriptor`.
`java.util`:	
`NI_TimeZone`	Functions that define the current timezone.

in terms of these "NI" classes. The grouping of native methods into classes is designed to make interposition more flexible (see Section 3.2.6).

3.2.2.3 Code control

Because class objects in Java define both object layout and the object methods, control over a child's process classes means the parent process can exercise control over what code is available in a child process, and with careful use of the language-level protection mechanisms, can "download" code into a child. The replacement of `java.lang.NI_Exit` in the previous section is one example.

3.2.3 Interprocess sharing

Despite the separate typespaces for each process and the extent to which a child typespace can differ from a parent's typespace, Alta still allows a limited amount of direct object sharing between two processes. Shared objects can be accessed directly in either process without indirection or other overhead. Initially, processes in Alta contain no visible shared objects;[2] the first visible shared object must be passed through IPC. Subsequent objects can be shared through the first object, or passed through IPC.

Safe sharing is possible because the virtual machine has complete information about the typespaces of any two processes that wish to communicate, and the virtual machine can mediate the initial communication between any two typespaces. The Alta virtual machine uses this information to guarantee that a shared object has equivalent class in both typespaces. Additionally, since a shared object effectively opens a communication channel between two processes, Alta must ensure that all potential objects that might be communicated through a shared object are also safe. Classes that pass these two tests—class equivalence and safe potential sharing—define the set of objects one process may share with another.

[2]There are many shared objects hidden within the kernel code.

3.2.3.1 Type equivalence in Alta

If an object were to be shared between two typespaces with inconsistent classes, the integrity of the virtual machine would be lost. For example, given the classes `BasicClass`, `KeyClassSafe`, and `KeyClassUnsafe` in Figure 3.3, consider two typespaces that resolve the name "BasicClass" to the same class file, `BasicClass`, but resolve the name "KeyClass" to `KeyClassSafe` and `KeyClassUnsafe` respectively. If an instance of `BasicClass` is shared between the two typespaces, then the `key` field would be treated differently in the two typespaces. The integer field of `KeyClassUnsafe` would be used in place of the object reference field in `KeyClassSafe`, which would allow forged pointers. The example demonstrates that the class of an object is not completely defined by the class file most closely associated with that object, specifically, the class of an instance of `BasicClass` is not fully described by the `BasicClass` class file.

Note that the inconsistency described in this example can never arise within

```
class BasicClass
{
        public KeyClass key;
        public String name;
}

final class KeyClassSafe
{
        private Object magicObject;
}

final class KeyClassUnsafe
{
        public int spoofMagic;
}
```

Figure 3.3. The simple classes `BasicClass`, `KeyClassSafe`, and `KeyClassUnsafe`. The class `BasicClass` is dependent on the resolution of the name `KeyClass`. To share an instance of `BasicClass` requires that the involved typespaces all resolve `KeyClass` and `String` to the same class files.

a single process (or a traditional JVM) because each name in a class file—in this example, the name "KeyClass"—is resolved only once. In Alta, however, the names referenced by a class file may be resolved differently in different processes.

To determine if an object can be shared between two processes, the virtual machine must determine if the *set* of class files that define the class of that object are equivalent in both processes. A class is defined by more than just a simple class file; a class is defined by the class of each field, the classes referenced in method signatures, the class of the superclass, and all of the implemented interface types. A class file only describes how a class is constructed by listing the names of the class file describing each of the relevant parts. Thus, two classes are equivalent if and only if all of the class files used to describe all of the parts of the class are equivalent.

A *single* class file can be defined as equivalent between two processes if it is derived from a single class file in a common ancestor process. More formally, identify a single class file as the pair $\langle N, P\rangle$, where P is the process in which N is the name of the class file, and define the notation $\langle N_c, C\rangle \underset{imm}{\Leftarrow} \langle N_p, P\rangle$[3] as: the class file $\langle N_c, C\rangle$ was resolved by the immediate parent of process C, process P, responding to the name N_c with the class file bound to N_p. A child process's class file, $\langle N_c, C\rangle$, and the parent process's class file, $\langle N_p, P\rangle$, from which it was derived are equivalent class files. More formally, this relation can be expressed as:

$$\langle N_c, Child\rangle = \langle N_p, Parent\rangle \text{ if } \langle N_c, Child\rangle \underset{imm}{\Leftarrow} \langle N_p, Parent\rangle.$$

This relation simply states that if a parent process replies to a child process's name request with a given class file then the class file is the same in the parent and child, regardless of the names they give it. Because this relation between parent and child process class files is an equivalence relation, it is reflexive, transitive and symmetric. The transitive closure of the relation defines a relationship $\Leftarrow$, where

$$\langle N_c, Child\rangle \Leftarrow \langle N_p, Ancestor\rangle \text{ if } \langle N_c, Child\rangle = \langle N_p, Ancestor\rangle.$$

[3] "$\Leftarrow$" could be pronounced "derived from."

Thus, for two distinct processes P_a and P_b the class files named N_a and N_b in respective processes are equivalent if they share a common ancestor class file. The resulting class file equality relation, = can be expressed as:

$$\langle N_a, P_a\rangle = \langle N_b, P_b\rangle \text{ if } (\langle N_a, P_a\rangle \Leftarrow \langle N', P'\rangle \text{ and } \langle N_b, P_b\rangle \Leftarrow \langle N', P'\rangle).$$

There are two aspects of Java which complicate this analysis: classloaders and interfaces. Classloaders were described in Section 3.2.2.1. Incorporating classloaders into the definition of a class file requires that the notion of a class name be extended. That is, a class file is actually identified, within a process, by the pair of name and classloader. Other than complicating the definition of a class name, this should have no impact on the definition of a class or on class equality. Interfaces are effectively abstract classes containing only abstract methods. Interfaces are treated as regular class file objects, and add no actual complications to the Alta type system.

Given this definition of class file equality, the class equivalence test in Alta is a simple recursive algorithm. Given an object to be passed through IPC, Alta determines the class of the object. Given the class in the source process's typespace, there must be an equivalent class in the destination typespace.[4] Starting with the basic class file, each referenced class file must also be equivalent in the two typespaces. This test determines if two classes are equivalent in disjoint typespaces. Referenced class names are followed until a base shareable class is encountered. Base shareable classes include primitive types—which are by definition equivalent in all processes—plus a select subset of the preloaded classes. See Table 3.3 for a complete list of the base shareable classes. Each of the base shareable classes is asserted to be safe.

[4] Alta currently does not handle loading a class when the first reference to it is from an object passed through IPC. Currently, if any required classes are unavailable in the target typespace, the object is not passed. Similar to page-faults during IPC in a traditional kernel, handling such a fault in Alta would require the kernel to generate a class-fault exception to the parent process. This, in turn, would require a thread to have two outstanding IPC operations at one time, which, in turn, requires Alta to support idempotent IPC operations, which it does not at this time.

Table 3.3. The base shareable classes.

Class	**Justification**
Primitives	All of the primitive types are, by definition, equivalent in all processes. (The primitive types are `boolean`, `byte`, `char`, `int`, `long`, `float`, and `double`.)
`java.lang.String`	A final class which is preloaded into all processes. Only contains references to `char[]` and `int`.
`java.io.NI_FileDescriptor`	A final class which is preloaded into all processes. Only references primitive types.
`edu.utah.npm.core.Cond`	A final class which is preloaded into all processes. Only references classes in the `edu.utah.npm.core` package.
`edu.utah.npm.core.Mutex`	A final class which is preloaded into all processes. Only references classes in the `edu.utah.npm.core` package.
`edu.utah.npm.core.PortSet`	A final class which is preloaded into all processes. Only references classes in the `edu.utah.npm.core` package.
`edu.utah.npm.core.ClassHandle`	A final class which is preloaded into all processes.
Arrays of shareable classes	Arrays of any shareable class are also shareable. For example, arrays of bytes, or arrays of `String`s are shareable.

For example, if an instance of the class BasicClass (see Figure 3.3 from the beginning of this section) is passed through IPC, Alta finds the class that defines BasicClass in each process. The BasicClass class file references three other classes, java.lang.Object (the implicit superclass), KeyClass and java.lang.-String. The base shareable classes, Object and String, are always equivalent in all processes, leaving KeyClass. The class file bound to the name KeyClass is looked up in each process. Next, assuming that class file is equivalent, all of the classes referenced in KeyClass are checked for equivalence.

Alta currently implements an even more restrictive subset of this algorithm. Alta only allows objects that are instances of base shareable classes or are instances of classes whose fields are all base shareable classes. This is all that has been necessary in practice, so far. If necessary, generalizing the implemented algorithm to the specified one should be straightforward. The only difficulty will be handling and tracking circular class file references in a class definition.

By separating the name of a class from its definition, Java's classes behave similarly to mixins. A mixin is an abstract subclass—a subclass whose superclass is not fixed [10]. The mixin may be applied to any appropriate superclass. This is similar behavior to a Java class in that the binding of the superclass to a class is done at run-time. In Java class flies, however, the name of the superclass is fixed in the class. In contrast with mixins, in Alta every name specified in a class file, e.g., the field types and in the method signatures, is bound at run-time.

3.2.3.2 Safe potential sharing

Simply having equivalent classes in two typespaces is insufficient for Alta to allow sharing. Because shared objects open up an unregulated communication channel between processes, Alta must ensure that arbitrary objects cannot be shared. For example, consider an object, X with a field "ref" of type java.lang.Object. If X is shared between two processes, then those processes may pass objects of any type through X by reading and writing from the "ref" field. Such arbitrary types may not be available or compatible in both typespaces, leading to corruption of the virtual machine. Even if all classes in the two typespaces are completely equivalent, in the face of dynamically loaded classes the virtual machine cannot predict if the typespaces will become inconsistent in the future.

The only object fields that the virtual machine can guarantee are safe are those that are not polymorphic or interfaces. For example, a field of class `java.lang.String` is allowable because the class is `final`—subclasses of `java.lang.String` are not allowed.[5] For non-`final` classes, the virtual machine cannot know what classes might be shared by as-yet-unknown subclasses. Therefore, the virtual machine cannot allow shared objects to contain fields of polymorphic classes (or interfaces). Note that references passed through IPC do not need to be instances of a `final` class because the virtual machine is dealing with a specific instance of the class.

It should be noted that the class `edu.utah.npm.core.ClassHandle`, a member of the base shareable classes, violates this rule. The class contains a field of class `java.lang.Class`, which is not a safe class. The implementation of `edu.utah.npm.core.ClassHandle`, however, is trusted and never allows the shared `Class` field to be visible in more than one process. This sort of analysis—gauging the visibility and safety of potentially unsafe fields—is only viable with trusted classes.

One more limit is placed on classes to allow them to be shareable: the classes must not contain static variables. This restriction allows the JIT compiler to inline static variable addresses. Because a shared object contains a method dispatch table pointer (also known as a vtable) the JIT code is shared with the object. To guarantee that this code is consistent with what would be generated in both processes, static variable references are disallowed. This is not a strict requirement. Relaxing it would imply that the JIT compiler be modified to support indirect access to static variables.

[5] Note that `final` is an overloaded term in Java: a `final` method is one that cannot be overridden in a subclass, a `final` field is one whose value cannot be changed, and a `final` class is one that cannot be subclassed.

3.2.3.3 Side effects of Alta's sharing model

One unusual side-effect of Alta's shared object model is that each process has its own `Class` object associated with each shared object. Thus locking an object's `Class` will not provide mutual exclusion across processes. This is important because methods that are both `static` and `synchronized` are defined to lock the associated `Class` object. This caveat, coupled with the restrictions imposed for safety, imply that arbitrary Java objects cannot be indiscriminately shared between processes. No existing Java class, however, is written to be executed in two different processes, so shared objects must be treated with some care in any event.

An additional side-effect of Alta's type model is that the binding between a class and its name is not fixed. This, combined with the fact that a single type can have two names in an Alta process, means that serialization of objects in a manner compatible to the JDK specification is difficult, as serialization depends heavily on representing a type by its name.

Direct object sharing poses problems for memory management and resource ownership. These problems are discussed in Section 3.2.5.

Despite these limitations and caveats, many objects are safely shared by the code in Alta's kernel. For example, cross-process capabilities are actually implemented as direct object references. Additionally, the IPCIF protocols (described later in Section 3.2.6) are able to pass many parameters as shared objects instead of marshaling their parameters into a byte buffer, copying the buffer and unmarshaling the parameters.

3.2.4 Core API implementation

The Alta core API (see Appendix A) provides the most basic set of services to Java applications. Its implementation is equivalent in many ways to the implementation of a traditional kernel. The Alta kernel multiplexes the underlying system to multiple processes. Just as in a traditional system, the kernel in Alta must protect itself from errant and malicious applications. The Alta kernel is fully preemptive.

3.2.4.1 Memory management in Alta

The majority of the nested process model core API is simple to support in Java since the kernel abstractions are entirely new to Java (e.g., ports, references, and processes) or easily match existing Java abstractions (e.g., threads). The only objects that are significantly difficult to provide in Java are the memory mapping objects.

In Fluke, the nested process model core API defines `regions` and `mappings` for mapping memory addresses from one process into another. Java has no language-level notion of memory addresses, and introducing such a notion merely to support these abstractions is unproductive. The IPC-based API in Fluke, however, defines a memory pool object that represents a much more abstract chunk of memory—effectively, just a size in bytes. In Alta, this higher level abstraction, the `MemPool`, is used to manage memory. When a new process is created, the parent process can create a `MemPool` to associate with that process. The memory pool is given an amount of memory, allocated from the current memory pool. When the process runs out of memory the process will make an IPC to the parent to invoke the garbage collector or request more memory.

While mempools are created and given a size by user-level processes, the details of the mempool are managed by the virtual machine; individual credits, debits, and availability checks are made by the virtual machine. Currently, Alta supports only a one-to-one mapping between processes and mempools. Supporting many-to-one and one-to-many relationships should be possible, but there is currently no motivation for such support. A future version of Alta, however, could support different types of memory, for example "wired" and "pageable" memory, and this distinction could be presented to applications through different mempools.

3.2.4.2 Maintaining kernel integrity

To maintain the integrity of the kernel's data structures in the face of arbitrary user code and user contexts, three basic issues must be confronted. All three involve unanticipated exceptions being thrown while a thread is executing kernel code. The kernel maintains a number of shared data structures that must be protected from inconsistencies that could be introduced if a thread were stopped in the middle of a kernel critical section.

First, the kernel must protect itself from `java.lang.Thread.destroy()` and from asynchronous exceptions thrown via `java.lang.Thread.stop()`. `Thread.destroy()` is a method on a thread object which stops the target thread dead in its tracks; `Thread.stop()` stops the target thread and throws an exception in that thread's context. The kernel protects itself by postponing stops and other interruptions when a thread enters the kernel. Postponed interrupts and stops are "posted" when the target thread exits the kernel. Traditional kernels use a similar tactic, only delivering signals to processes while they are executing in user mode [33, p. 97]. In Alta, the transition from "user mode" to "kernel mode" is explicit in the code by use of the kernel-private methods `Thread.startSystemCode()` and `Thread.endSystemCode()`.

The second issue facing the Alta kernel is running out of stack space during execution. This poses the same problems as interruption, but stack overflows cannot simply be "postponed." Upon entry to system methods, the available stack space is checked against a predefined limit, analogous to traditional, hardware-based kernels that run kernel code on a separate stack of fixed size that is "known" to be sufficient (usually 4k or 8k bytes). The stack size check is not yet implemented in Alta.

The final and most complicated issue the core Alta code must deal with is running out of memory while executing kernel code. Like stack overflows, out of memory conditions cannot be "postponed." Unlike entry-time stack checks, a check for sufficient memory at kernel entry time is insufficient as other threads in the system may use the memory before the in-kernel thread needs it. Alta approaches this problem by pushing as much memory allocation out of the kernel as possible. In fact there is no explicit object allocation in the Java portion of the Alta kernel. All of the system calls operate on state provided exclusively by user mode code. In addition to avoiding out of memory conditions within the kernel, this increases the precision of Alta's resource accounting. For example, when objects such as a port or thread are created by an application, all of the required kernel state is allocated and initialized—in user-space. In a traditional hardware-based system, kernel state and application state are usually quite separate. For example, a file descriptor in Unix is user-mode state associated with some separately allocated kernel state. Traditional kernels cannot allow the user to allocate kernel data structures. Alta, on the other hand, can take advantage of the fine-grained access control provided by Java's type system to attach private kernel state to application-visible objects. Figure 3.4 shows how the `edu.utah.npm.core.Space` object, which an application creates when starting a new process, creates the associated kernel state at the same time. The vast majority of the operations on the kernel objects (listed in Figure 2.1) never make allocations; they operate entirely on existing objects. For those methods that do require temporary objects, the objects can be allocated by the method before entering kernel mode. Figure 3.5 shows a fragment of the `Reference.check()` method which checks the integrity of a `Reference` object. This method creates a temporary `Link` object (for a copy of the `Reference`-internal `Link` object that represents the actual object reference). Because the entry to kernel mode is explicit

```
public final class Space
{
        // A kernel-internal cross-process reference
        private final Link keeperPortLink = new Link();

        // For the queue of threads active in this process
        private Thread s_threadQHead_ = null;
        private Thread s_threadQTail_ = null;

        // An internal rendezvous object for threads
        // in this process
        final Object stopCond_ = new Object();

        // ...
}
```

Figure 3.4. A fragment of the `edu.utah.npm.core.Space` class implementation showing what kernel state is allocated and initialized when the object is created.

```
// In the Reference object
public boolean check()
{
        Refable obj = null;
        boolean isActive = false;
        final Link linkCopy;

        linkCopy = new Link();          // allocate outside
                                        // kernel-mode

        Thread.startSystemCode();       // Enter kernel-mode

        // do the actual check
        ...

        Thread.endSystemCode();         // Exit kernel-mode
}
```

Figure 3.5. The `check()` method of `edu.utah.npm.core.Reference` performs memory allocations before entering system code. This separation is possible because a type-safe language based operating system can safely decouple kernel entry from entry to kernel mode.

in Alta (through `Thread.startSystemCode()`), the temporary object allocation can be done inside the kernel but outside kernel mode. Thus if the allocation of the temporary object fails, a standard `OutOfMemoryError` will be thrown without disrupting the kernel.

There are still areas of Alta's kernel that can trigger out of memory exceptions. First, verification and JIT compilation of kernel methods cause the virtual machine to allocate memory. These allocations could be avoided by preverifying and precompiling all of the kernel methods when a process is created. Second, the virtual machine dynamically allocates monitor locks as they are required. Since the kernel itself uses very few monitor locks, those that are used could be preallocated for the required kernel objects. These changes would make Alta's kernel completely allocation-free and would eliminate a whole class of errors and race conditions related to memory allocation in the kernel.

3.2.5 Core resource controls

Processes in Alta are subject to comprehensive memory controls enforced by the virtual machine. Every allocation made by a thread is charged to the memory pool associated with that thread's `Space`. This accounting includes buffers to hold compiled bytecode, the per-space typespace map, every Java object allocated by a thread, etc. Every object in Alta has an owning `MemPool` associated with it. The `MemPool` is credited when the garbage collector returns an unreachable object back to into the pool of available memory. The garbage collector introduces a delay between releasing an object and reclaiming the memory used by the object. Type-safety constrains the system to never allow dangling pointers. Together these two factors constrain the system's ability to terminate a process and reclaim all of its memory: to completely reclaim all of a process's memory, none of the objects in that process can be reachable and the garbage collector must be invoked to reclaim the memory.

3.2.5.1 Memory accounting and shared objects

Resource accounting and control are complicated by the sharing of objects [29]. In traditional hardware-based systems, a page of memory can simply be revoked by the operating system: any processes that try to access that memory will fail. For example, if a process is killed by the system, all of its pages can be unmapped and reused. In a type-safe system, an object cannot simply be unmapped and reused if there are existing references, as the existing references would invite type-safety violations. In a hardware-based system, revocation can lead to the corruption of a single process, but in a language-based system type-safety violations can lead to the corruption of the entire system.

Given a process, `A`, that "owns" objects that are shared with another process, termination of the process `A` will not result in the reclamation of all of its memory as the other process will effectively keep that memory reachable. Alta allows object sharing because the hierarchy of the nested process model can be used to contain sharing in a natural and efficient manner. A processes in Alta can tightly contain its child processes or let them communicate efficiently via shared objects. In this way, the application can trade the ability to cleanly terminate a child process for more efficient communication with that process.

Figure 3.6 diagrams the three different types of interprocess sharing that can occur in the nested process model. In Figure 3.6(A), a parent process allocated an object and the child process has a reference to that object. In terms of process termination, this sort of sharing is harmless. If the parent terminates the child, all objects are reclaimed. If the parent is terminated, the child is necessarily terminated also. Additionally, this sharing paradigm is very useful: it is the standard server model, where the lifetime and usefulness of a server object depends on the lifetime of a client's references to that object. Of course, in this case, the server has lost the ability to reclaim the object unless it can convince the child to drop its reference or the child is terminated.

Figure 3.6(B) presents the case of a parent holding a reference to an object in a child process. Again, in terms of termination, this sort of sharing presents no significant problem. If the parent process terminates the child, the object is simply promoted to the parent—the parent effectively owns the whole child to begin with. Again, in this scenario, if the parent is terminated, the child is necessarily terminated also. This sharing format is, again, useful: a child could, for example, pass a reference to a local buffer to a parent server, which would be able to fill the buffer directly without copying.

Finally, Figure 3.6(C) diagrams sibling processes sharing an object. Communication with a sibling process requires the cooperation of the parent as it is the first common communication endpoint. In terms of termination, by allowing two sibling processes to share objects directly, a parent process has linked the clean termination of those processes; in order for the parent to terminate one child process and reclaim *all* of its memory, the parent must necessarily terminate the other child process, too. In Alta the allocator of an object is charged. This tradeoff of clean termination versus fast communication is left to the discretion of processes in the system. Alta

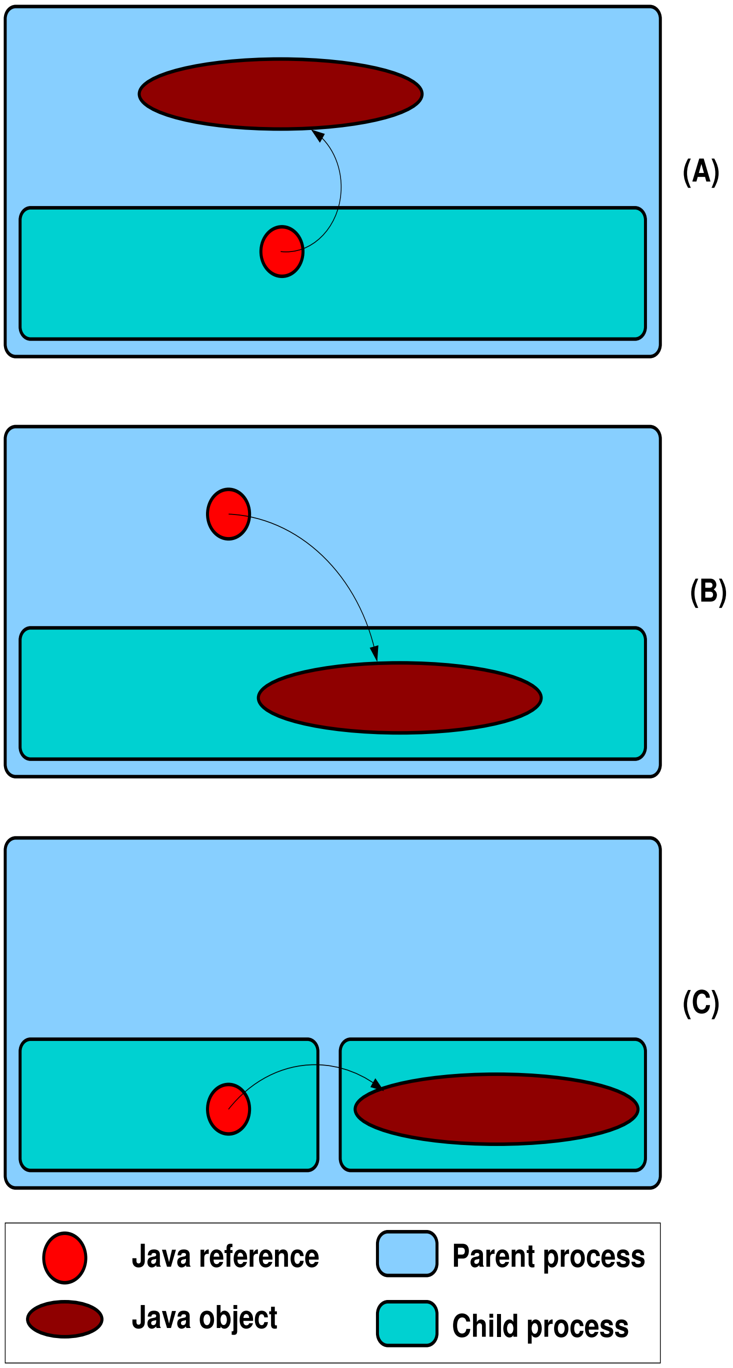

Figure 3.6. The three different modes of interprocess sharing in the nested process model. (A) represents a child holding a reference to an object allocated by its parent. (B) represents a parent holding a reference to an object in its child and (C) represents a process holding a reference to an object in a sibling.

provides sufficient infrastructure for any process to implement these policies over the processes it is managing.

3.2.6 IPC-based interfaces

The Alta IPC-based interfaces, or IPCIF (see Appendix B), provide applications running on Alta with an interface to higher-level services than provided by the kernel. Except for the Parent interface (Section B.1), the IPCIF API relies on the kernel only for the communication mechanism; the protocols are otherwise independent of the kernel.

In Fluke, the IPC-based interfaces support the POSIX orientation of the standard libraries Fluke applications are linked against. In Alta, however, the IPCIF are designed to support the Java standard libraries and are based around the native interfaces upon which those libraries depend (see Table 3.2). Despite the different focus, the interfaces are quite similar. Both systems have interfaces for accessing the file system—for example, opening a file or deleting a file—an interface for reading and writing individual files, and a memory management interface for requesting and returning large blocks of memory. Alta has an interface for memory related operations such as invoking the garbage collector and querying the amount of memory available. In both Alta and Fluke the *Parent* interface is the only interface to which a child process initially has access. This interface serves as the most basic name service, providing access to other interfaces. While the Parent interface is directly supported by the kernel (the Alta kernel invokes class name resolution requests on the parent interface directly), the other interfaces are completely independent of the kernel. This separation means that IPC-based interfaces can be tailored for specific environments or to support new server interfaces without changing the kernel.

Alta does not support the Abstract Window Toolkit (AWT) libraries at this time because the Kore library—the standard Java library implementation Alta uses—does not have working AWT support. Unlike the other IPCIF interfaces, the division into "server" and "client" components for the AWT has no analogue in Fluke. The best example of an IPC-based GUI is the X windowing system, so an AWT interface on IPCIF should be straightforward. The Kore library is described later, in Section 4.1.1.2.

3.2.7 Exportable state

One of the basic tenets of Fluke is that all state of all objects is extractable at all times [21, 46]. Actually, in Fluke, much of the associated kernel state is not exported when a kernel object is extracted. All that is needed to "extract" an object is enough information to correctly recreate the object later. Additionally, for many of the core kernel objects in Fluke, state is only extracted when the object is not in the "middle" of an operation. For example, the state of a thread object is extractable only if the thread is not executing any kernel functions. If the thread happens to be executing a kernel function when its state is extracted, the kernel functions are canceled and undone, and then the state of the thread is extracted [21].

While wholesale state extraction is useful for writing a comprehensive checkpointer that can take a "snapshot" of a set of related processes [46], and for writing a virtual memory nester that can write a process's kernel state out of core, state extraction is not useful for resource control and process management. It is important to note that services such as virtual memory and checkpointing are not impossible in Alta, but they must be implemented within the virtual machine.

The largest stumbling block to implementing state extraction in Alta would be the safe restoration of an exported thread's stack. A representation of a thread stack would have to be developed as well as a mechanism for verifying that each frame on the externalized stack matched the requirements of the associated method. Given that an externalized stack could be manipulated, the ability to "download" code into a child process would be jeopardized as a process could invoke trusted code, stop the thread, extract the thread's stack, manipulate the stack to a new state, and restore the thread, thus bypassing whatever checks may exists in the trusted code. Together these restrictions on complete state extraction outweighed the benefits and Alta was not designed to include state extraction.

3.3 Conclusion

Starting with the nested process model, as implemented in Fluke, Alta was created by mapping the model into a Java virtual machine. Alta gives each process the impression of running on its own Java virtual machine, yet still allows processes to directly share many types of objects. A process in Alta may control the types visible in child processes, and may maintain resource limits on those child processes. Finally, Alta uses a number of standard operating system techniques for maintaining control over user processes.

CHAPTER 4

IMPLEMENTATION AND ANALYSIS OF ALTA

This chapter demonstrates some simple examples of using Alta, presents a quantitative analysis of Alta's implementation in terms of performance and relative size, and finally compares Alta and Fluke.

4.1 Implementation

4.1.1 Infrastructure

The Alta virtual machine supports the basic features required of a Java virtual machine such as garbage collection, threading, JIT compilation, bytecode interpretation, verification, and all of the features of the core Java libraries. The majority of these features were implemented by the software Alta is built upon: the Kaffe virtual machine and the Kore standard libraries.

4.1.1.1 Kaffe

The Alta virtual machine is based on Kaffe [51], a freely available virtual machine developed by Tim Wilkinson. Alta is based on Kaffe version 0.9.2, which supports most of the features of JDK 1.0.2 and used Sun's JDK 1.1.3 class libraries. For Alta, a large number of features introduced into later versions of Kaffe–such as an improved threading infrastructure and a new garbage collector—were ported back to Alta.

4.1.1.2 Kore

The core Java libraries for Alta are based on the implementation provided by the Kore project [13]. Kore is a clean-room implementation of the Java core libraries and is compatible with JDK 1.0. Kore provides a very clean and minimal interface between the Java libraries and the native code they depend upon.

4.1.2 Implementation Statistics

The Alta implementation is reasonably small, both in terms of the changes made to the virtual machine (several thousand lines of code) and the runtime support code (several hundred lines of code).

The implementation of Alta is easily broken into two pieces—a Java component and a C component. The Java component consists of the core Alta libraries, which implement the basic NPM primitive objects. Also in the Java component are the IPCIF libraries that provide the high-level services on top of IPC. The core library is approximately three thousand lines of code (three kloc), while the IPCIF library is less than two kloc.[1] As a point of reference, the Kore library, which provides standard Java libraries compatible with JDK 1.0.2 but without AWT support, is about four kloc, while the JDK 1.1.7-compliant standard Java libraries for Kaffe v1.0b2 without including AWT support is just under nine kloc.

The Kaffe virtual machine (version 0.9.2, on which Alta is built) contains approximately ten kloc of C source, while the Alta virtual machine weighs in at thirteen kloc of C source. Probably one-half of that increase is code that fixes bugs, adds significant debugging infrastructure or supports new features from later versions of Kaffe. For example, a new threading infrastructure was incorporated into Alta from a later version of Kaffe. Additionally, Alta, unlike Kaffe, provides native methods for the Kore libraries.

In terms of binary size, on FreeBSD 2.2.6 the Alta virtual machine binary is approximately 350K statically linked or 230K if the standard FreeBSD libraries are dynamically linked. A Kaffe version 0.9.2 binary is 350K statically linked or 200K dynamically linked. The 29 core Alta class files, i.e., the `edu.utah.npm.core.*` package, use 53K of disk space, the 28 IPCIF class files use 55K, while the Kore

[1] A "line of code" is a nonblank, noncomment, line of program source that contains more than just punctuation or preprocessor directives. For all of the listed packages this line of code count is between 30% and 35% of the raw lines of text.

library's 163 files use 183K. In comparison, the JDK 1.1.7-compliant standard Java libraries for Kaffe v1.0b2 contains 532 class files and requires about 915K of disk space and the Sun JDK 1.1.6 core runtime (all of the `java.*` packages) contains 721 class files that require 1.6 megabytes of disk space.

Overall, Alta and its libraries are moderately sized, and the changes made to the virtual machine are not overwhelming.

4.1.3 Implementation Lessons

Java is a simpler and safer language than C. The Java compiler catches far more errors than any C compiler. On the other hand, the high-level nature of many of the basic Java abstractions occasionally hampers the system programmer. For example, because `java.lang.String` objects are real Java objects, using them in critical parts of the kernel (e.g., for debugging) can be troublesome. Specifically, the class name resolution code that performs an IPC to a parent process cannot print a String because the compilation of `String` methods may cause recursive entry into the IPC class name resolution code. One other example of the lack of system-level control while programming in Java is evident when allocating temporary objects. In C, a `struct` local variable is allocated on the stack, but there is no such analogue in Java: all Java objects are dynamically allocated on the heap. Stack allocations are a fast and simple way of avoiding the overhead and errors associated with dynamic memory allocation and such techniques can be critical in a kernel. In Alta, only one allocation in the kernel needed to avoid the heap allocator. In C this site would have used stack allocation; in Alta this need was addressed by dynamically allocating the object in the context of the caller—avoiding the heap allocator in the critical code. Just as C programmers can delve into assembly for truly low-level control over a system, a Java systems programmer always has the option of using C code for implementing constructs that are awkward or inefficient in Java.

4.2 Using Alta

Alta is invoked in the same manner as other Java virtual machines. The only required argument to Alta is the name of the initial class to load. Optional arguments can be provided before the class name. Arguments given after the class name are arguments to the `main()` method of the initial class. Class files are found on the local file system via the `CLASSPATH` environment variable.

Here is the traditional "Hello World!" example for Alta:

```
% alta HelloWorld
Hello World!
```

Alta loads the class file `HelloWorld.class` and executes the static `main()` method in `HelloWorld` which simply prints the string "Hello World!" to standard output.

The next example demonstrates how to run HelloWorld as a subprocess of the Alta sandbox. The Alta sandbox was described in section 1.1.

```
% alta edu.utah.npm.apps.sandbox.Main HelloWorld
Hello World!
```

In this example, "HelloWorld" is the name of the application the sandbox starts. The sandbox is an Alta application that creates a child process and optionally regulates the child's access to classes, the file system, etc. In this example, the sandbox only interposes on class name resolution but does not limit file system access or memory usage.

As a more complex example, the following fragment is a trace of `javac`, the Sun Java compiler, in a sandbox. The compiler compiles the Java source file `HelloWorld.java` into the file `HelloWorld.class`.

```
% alta edu.utah.npm.apps.sandbox.Main \
        sun.tools.javac.Main -verbose HelloWorld.java
[parsed HelloWorld.java in 314ms]
[loaded /lib/classes.zip(java/lang/Object.class) in 39ms]
[checking class HelloWorld]
[loaded /lib/classes.zip(java/lang/String.class) in 20ms]
[loaded /lib/classes.zip(java/lang/System.class) in 14ms]
[loaded /lib/classes.zip(java/io/PrintStream.class) in 10ms]
[wrote HelloWorld.class]
[done in 965ms]
```

Note that the `-verbose` flag is passed to javac and causes javac to print out the time required to perform various steps of the compilation. Again, the sandbox nester is not interposing on any interfaces.

The next example demonstrates that the sandbox application can be nested, creating a trivial hierarchy of nested processes. This is feasible because the sandbox nester relies on the same interfaces that it exports.

```
% alta edu.utah.npm.apps.sandbox.Main \
        edu.utah.npm.apps.sandbox.Main HelloWorld
HelloWorld!
```

The final example demonstrates the result of running the java compiler with a restricted memory pool. The `-memlimit` argument to the Alta sandbox specifies the size of the memory pool for the compiler. Because the limit given is less than the working size of the compiler, it runs out of memory and is cleanly killed by the sandbox.

Note that each time the child runs out of memory, it faults to the sandbox. The sandbox never increases the size of the child's mempool, it simply prints a message and invokes the system-wide garbage collector. When the size of the child's working set exceeds the memory pool, the sandbox nester terminates the child.

```
% alta edu.utah.npm.apps.sandbox.Main -memlimit 2m \
        sun.tools.javac.Main -verbose HelloWorld.java
[parsed HelloWorld.java in 316ms]
Child ran out of memory (short by 9496)
[loaded /lib/classes.zip(java/lang/Object.class) in 144ms]
Child ran out of memory (short by 12656)
[checking class HelloWorld]
[loaded /lib/classes.zip(java/lang/String.class) in 20ms]
Child ran out of memory (short by 1080)
Child ran out of memory (short by 152)
[loaded /lib/classes.zip(java/lang/System.class) in 118ms]
Child ran out of memory (short by 344)
Child ran out of memory (short by 168)
[loaded /lib/classes.zip(java/io/PrintStream.class) in 114ms]
Child ran out of memory (short by 1568)
Child ran out of memory (short by 9536)
Child ran out of memory (short by 152)
Child ran out of memory (short by 176)
```

```
Child ran out of memory (short by 8744)
Child ran out of memory (short by 5416)
Child ran out of memory (short by 1432)
Child ran out of memory (short by 1432)
Working set exceeds memory limit.  Child terminated.
```

4.3 Performance evaluation

I compare the performance of four different virtual machines: the Alta virtual machine, two "stock" versions of Kaffe, and a Microsoft Java virtual machine. The Alta virtual machine is based on Kaffe v0.92 but incorporates many of the changes that brought Kaffe to v1.0b2, thus the Alta virtual machine lies somewhere "between" these two stock Kaffe virtual machines. Alta uses the Kore class libraries while Kaffe v0.92 uses Sun's JDK 1.1.3 class libraries. Kaffe v1.0b2 uses its own custom set of class libraries; as does the Microsoft virtual machine. Benchmarks where the differences between the class libraries are likely to make a performance impact are so noted. Table 4.1 shows each virtual machine, its libraries, and approximate JDK version. Note that for the Kaffe and Kore libraries compliance with the given JDK version is not complete.

All results reported in this chapter were obtained on a 300Mhz Intel Pentium II CPU with 128Mb of main memory and a 512K L2 cache. This architecture includes a 16K L1 instruction cache and a 16K L1 data cache. The system was running an installation of FreeBSD version 2.2.6, with only the necessary system services active. The Microsoft JVM results were obtained on an identical machine running Windows 95 using the Microsoft Java virtual machine shipped with Visual

Table 4.1. Java virtual machines, standard Java libraries, and JDK version.

Virtual Machine	**Libraries**	**"version"**
Alta	Kore	JDK 1.0.x
Kaffe v0.92	Sun	JDK 1.1.3
Kaffe v1.0b2	Kaffe	JDK 1.1.x
MS JVM	Microsoft	JDK 1.1.7

J++ v6.0.

All of the Java virtual machines measured in this chapter support JIT compilation. When appropriate, tests were written to avoid including the time used by the JIT compiler in the results. For tests that measure whole-application performance, JIT compilation was necessarily included. Additionally, garbage collection was prevented during timing loops by cleaning the heap before a timing loop and ensuring that sufficient space for dynamic allocation requests was available.

For Alta, CPU usage is measured with the Pentium cycle counter.[2] The cycle counter provides a very accurate, low-overhead measure of processor time. It is directly accessible in a processor register from user-mode. For the stock versions of Kaffe, the standard method `java.lang.System.currentTimeMillis()` was modified to return the current value of the cycle counter instead of the current time in milliseconds. For the Microsoft virtual machine, the "PIT timer" was accessed through the Win32 kernel function `QueryPerformanceCounter()`. This timer fires 1,193,180 times per second. That translates to just over 251 cycles per PIT clock cycle on a 300Mhz machine. The Microsoft virtual machine numbers are reported as a number of clock cycles derived from the measured number of PIT ticks.

4.3.1 Benchmarks

I present performance benchmarks of a number of distinct features of Alta: object allocation, thread-switching, IPC, process creation, and file-system read/write. For those features that are available in a basic Java virtual machine, performance is compared to stock versions of Kaffe and to a Microsoft Java virtual machine.

These comparisons serve two purposes. First, they compare the Kaffe-based virtual machines to the Microsoft virtual machine, an aggressively optimizing virtual machine [48, 54], highlighting areas where Alta and Kaffe could be improved in general. Second, by comparing with the base versions of Kaffe, these microbenchmarks provide insight into the cost of supporting multiple processes in Alta,

[2] The installed version of FreeBSD was modified to not reset the cycle counter at each timer interrupt, as is done by default.

with respect to small, specific operations.

4.3.1.1 Basic virtual machine features

The first set of benchmarks measure basic features of a virtual machine: assignment, run-time type checking, and method invocation. Each operation was repeated (10,000 times) and the total cost of the experiment was divided by 10,000. Results from 34 of these trials were averaged (and rounded to the nearest whole cycle) to get the numbers shown in Table 4.2. For all of the results in this table, the range covered by the thirty-four trials was never more than eight cycles. Architectural artifacts such as cache effects, branch target alignment, mispredictions and memory stalls can easily generate timing variations on the order of 15 cycles, so results and deviations less than about fifteen cycles should simply be read as "real small."

Two conclusions can be drawn from these results. The first, and most important, is that the performance of basic features of the virtual machine are not significantly changed by supporting multiple nested processes. The two discrepancies are assignment to an `Object` array (the fourth line of the table) and synchronized

Table 4.2. Basic benchmark results for the four Java virtual machines.

Benchmark	**Alta**	**Kaffe v0.92**	**Kaffe v1.0b2**	**MS JVM**
(Do nothing)	7	6	8	2
`o = null`	8	9	7	3
`int[]` assign	13	12	13	5
`Object[]` assign	34	28	28	17
`instanceof` (same)	57	49	28	*N/A*
`instanceof` (different)	91	97	39	*N/A*
checkcast (same)	59	65	41	*N/A*
static method (this)	6	5	5	2
static method	5	7	5	3
instance method	24	24	24	8
instance method (args)	35	34	26	12
synchronized method	156	132	250	47

Times reported in cycles.

method invocation (the last line of the table). Assignment of an object into an `Object` array is slower in Alta due to the changes made to the type system. These changes also impact the "`instanceof` (same)" benchmark, which measures the cost of Java's `instanceof` operator. Both benchmarks include a runtime type check; because Alta has a slightly more complex notion of type than a traditional Java virtual machine, Alta takes slightly longer. The synchronized method invocation benchmark slows considerably in the later version of Kaffe due to changes in the locking primitives. Note that the 1.0b2 version of Kaffe includes optimized method invocation and optimized runtime type checking.

The second conclusion that can be drawn from these results is that Alta is substantially slower than the Microsoft virtual machine. Compared to the Microsoft virtual machine, the Kaffe-based virtual machines take three times as long to invoke instance methods, three times longer to invoke synchronized methods, and about twice as along to do everything else. This result implies that the absolute performance of the Kaffe-based virtual machines could be dramatically improved by optimizing the most basic primitives.

Continuing with benchmarks of basic virtual machine operations, Table 4.3 presents three measurements of thread overhead. The first row presents the cost of switching threads using `Object.wait()` and `Object.notify()`. The second row presents the cost of switching threads using the `Thread.yield()` call. Both of these tests create optimal conditions: the threads do nothing but switch back and forth. The wait/notify test measures thread switching overhead in the case of two

Table 4.3. Thread switching and thread start costs.

Benchmark	**Alta**	**Kaffe v0.92**	**Kaffe v1.0b2**	**MS JVM**
Thread switch (wait/notify)	540	1,707	1,450	13,881
Thread switch (yield)	186	*BUG*	467	6,711
Thread start	55,413	93,209	49,021	222,308

Times reported in cycles.

threads that are trying to synchronize with each other. The yield test measures thread switching overhead for the case of threads that are simply sharing the same processor.

The third row of Table 4.3 shows the time taken to start a new thread. This test measures the time from the invocation of `Thread.start()` to the beginning of the `run()` method in the target thread—it does not include the time to allocate the thread object. It should be noted that the locking and thread switching in Alta have been optimized with aggressive inlining and lock caching and the results for Kaffe v1.0b2 could be trivially brought in line with Alta's results as both virtual machines use the same threading package. (Kaffe v0.92 on the other hand, uses a completely different threading package.)

Kaffe's thread support is written as a user-mode thread package implemented with `sigsetjmp()` and `siglongjmp()`, while the Microsoft virtual machine uses kernel threads. Comparing a user-mode implementation to a kernel-mode implementation is usually a comparison of "apples to oranges," but Kaffe's thread package correctly uses nonblocking I/O operations to mitigate most of the shortcomings of a user-mode thread package, which eliminates the major distinction between user-level and kernel-supported threads. On the other hand, by using a user-mode thread implementation, Kaffe cannot simultaneously schedule threads on multiple CPUs.

These benchmarks show that support for nested processes has not impacted the thread scheduling primitives of the Java virtual machine. It must be noted, however, that the nested process model's thread scheduling model, CPU inheritance scheduling, has not been implemented in Alta.

4.3.1.2 Object allocation

The next set of benchmarks measure the time to allocate an object. Alta adds a single 4-byte pointer to the per-object overhead and charges each allocation against the appropriate memory pool. To better quantify this overhead, additional Alta results are obtained with memory pool support completely removed—that is, Alta was compiled without the per-object overhead and without the memory accounting code included. Like the threading support, object allocation in Alta has been optimized—mostly through aggressive inlining. For comparisons to Kaffe, Alta's GC system is closer in lineage to the later version of Kaffe.

As shown in the first two columns of Table 4.4, the per-object accounting scheme in Alta adds an overhead of 50 to 150 cycles to the allocation cost of a basic object. There is significant overhead for the allocation of the kernel-managed core objects when memory pool support is enabled, as seen in the rows for the core `Reference()`, `Cond()`, `Mutex()`, and `Thread()`. For these objects, the overhead of memory pool support is approximately 2500 cycles (about $8us$ on a 300Mhz machine). These extra cycles are used to register the core object with the owning memory pool so that when the memory pool is destroyed the core objects will be properly cleaned up and destroyed. Note that `IPCPayload` and `ClassHandle` are not "core objects," they only exist to simplify passing parameters to and from some system calls.

While there is a much more substantial overhead for core objects, these objects are generally allocated only when starting up new processes or threads. Addition-

Table 4.4. Cost of creating a variety of objects.

Allocation	**Alta**	**Alta (-MP)**	**Kaffe v0.92**	**Kaffe v1.0b2**	**MS JVM**
(Nothing)	8	8	8	9	10
(`o = null`)	9	9	8	10	8
`java.lang.Object()`	396	324	534	673	145
`int[32]`	680	613	952	974	405
`SubClass()`	475	373	646	735	145
`SubClass(1,2,3)`	493	355	658	725	155
`java.lang.Throwable()`	1,111	985	*BUG*	2,053	13,441
core IPCPayload()	707	432	-	-	-
core ClassHandle()	409	341	-	-	-
core Reference()	3,573	1,734	-	-	-
core Cond()	5,046	2,537	-	-	-
core Mutex()	5,883	3,083	-	-	-
core Thread()	11,861	8,695	-	-	-

Times reported in cycles.

ally, the code that registers core Alta objects is written in Java and would benefit from improvements to bytecode compilation.

To reduce the overhead (both in time and space) of charging each object to a memory pool, a virtual machine could charge a memory pool on a per-page basis and allocate multiple objects out of the page. This approach has the advantage of reducing per-allocation costs by amortizing them over the number of objects in a page, but has the disadvantage of increasing the memory usage of an application because the application would be required to pay for an entire page of objects, even if only one or two objects are allocated on that page. Despite these shortcomings, this approach would probably bring an improvement to the per-object accounting overhead. This cost is compounded in virtual machines, such as Kaffe, that pre-divide pages into similar sized objects. Notice that this optimization would not greatly reduce the overhead associated with Alta's kernel objects.

In summary, while there is a measurable overhead to per-object accounting, it is insignificant for basic objects. For kernel objects there is a more significant cost.

4.3.1.3 Interprocess Communication

This section provides a breakdown of the interprocess communication (IPC) costs in Alta. The single number that all the other results in this section revolve around is 18,524 cycles (just under $62us$ on a 300Mhz machine): the time required for a complete, local, null IPC. Specifically, finding a server thread from a port reference, connecting to the server, sending a null request and receiving a null reply and then disconnecting. To put this measurement in perspective, a null IPC in Alta is almost eight hundred times more expensive than a method invocation. Like all of the IPC results in this section, this number is the average of 34 trials. Each trial makes 10,000 repeated null IPCs and divides the time taken by the number of iterations. The standard deviation of the 34 null IPC trials is 83 cycles.

As a point of comparison, a Fluke kernel with the "same" IPC path completes a null IPC in about 7,500 cycles. A more detailed comparison between Alta and Fluke is made in Section 4.4. The most recent incarnation of Fluke, which contains a completely rewritten IPC path can complete a null IPC in about 3,800 cycles.

In addition to the null IPC benchmark, I present results for copying data through IPC, sharing objects through IPC and a breakdown of costs along the IPC path.

4.3.1.3.1 IPC with data. Two slightly more complex IPC benchmarks are the time to marshal, send, receive and unmarshal 3 integers and the time to marshal, send, receive and unmarshal a 100-byte `java.lang.String`. Alta requires about 22,000 cycles to send three integers, implying that it takes on the order of 3,500 cycles to marshal three 32-bit integers into a byte buffer, copy the byte buffer from client to server, and then unmarshal the three integers from the byte buffer. Transferring a 100 character String through IPC requires about 31,000 cycles, implying that an astounding 12,500 cycles are required to convert a String to an array of 100 bytes, copy the 100 bytes, and convert them back to a String.

To verify this overhead, two benchmarks that attempt to measure the non-IPC computations were run on all of the virtual machines. The first benchmark marshals three integers into a 12-byte array, copies the byte array, and unmarshals the three integers out of the array. The second benchmark marshals a 100-byte String into a 100-element byte array, copies the byte array, and unmarshals a String from the resulting byte array. Table 4.5 displays the results. Note that all four virtual machines use a different class library, and much of the difference in the String benchmark could be due to the performance of the String class's implementation. The Microsoft virtual machine is probably optimizing the integer constants out of the loop in the "Integer marshal" benchmark and might be inlining the array copy in both benchmarks. The discrepancy between the numbers in this table and the

Table 4.5. Marshaling costs for three integers or a 100-byte String.

Benchmark	**Alta**	**Kaffe v0.92**	**Kaffe v1.0b2**	**MS JVM**
Integer marshal	677	518	530	38
String marshal	8,017	6,925	9,399	3,637

Times reported in cycles.

estimates in the previous paragraph is quite noticeable. There are 3,000 "missing" cycles for the integer benchmark and 4,500 "missing" cycles in the String-copy benchmark.

Notice that the performance gap between the Kaffe-based virtual machines and the optimizing Microsoft virtual machine has increased relative to the difference in previous benchmarks. More Java code is used in the benchmarks, so the Microsoft optimizer has more code to work with.

4.3.1.3.2 Sharing via IPC. Another way to send a 100-character `java.lang.String` is to send an object reference through IPC. Doing so with Alta requires about 20,000 cycles—1,500 more cycles than a null IPC, but about 11,000 less than marshaling and unmarshaling the String data.

The benefit of copying or sharing smaller data items is overshadowed by the relatively high cost of IPC in Alta. If Alta's IPC logic were significantly faster, the cost of a data copy (which is basically fixed by the processor) would be a more significant contribution to overall IPC time, and thus sharing data through IPC would be more appealing (as IPC will always have a nonzero cost). Still, the advantages of sharing a complex object and thus avoiding IPCs will always be appealing. For example, a shared object representing an open file could correctly keep track of the number of bytes available in the file and entirely avoid an IPC on `available()` calls.

4.3.1.3.3 IPC cost breakdown. Table 4.6 shows a breakdown of where time is spent in a null, round-trip IPC. The first column identifies the stage of the IPC; the next two columns show the timestamp, in cycles, from the beginning of the sequence and the difference from the last step, respectively. The last column describes the state the thread is in and what it is doing. The entries in bold font are the obvious bottleneck candidates. Additionally, the absolute times in this table include the act of recording timestamps in the critical path, and thus the overall execution time is inflated. Specifically, before exiting an IPC system call there is at least an additional 2,500 cycles of overhead to write back the set of timestamps from the IPC call; this overhead is only explicitly recorded in step 7 where the

Table 4.6. Per-stage costs of a complete, round-trip null IPC.

	Time	Diff.	Description
1	0	0	Client thread entered main IPC function.
2	374	+374	Client thread will start “connect” phase.
3	796	+422	Client cleared empty server link.
4	**2,558**	**+1,762**	Client found the waiting server thread and captured it.
5	3,503	+945	Client transferred null request to the server thread.
6	3,798	+295	Client will begin reversing connection.
7	**10,334**	**+6,536**	Server thread has awoken and will return to user mode (it has completed the previous ack-send-wait-receive) and it will immediately begin a new ack-send-wait-receive.
8	**15,246**	**˜1,700 (+4,912)**	Server entered main IPC function and will start an ack-send-wait-receive. Approximately 3,200 cycles of extraneous measurement overhead were factored out.
9	15,453	+207	Server thread has entered main IPC loop.
10	15,696	+243	Server is about to begin handling the “ack-send” phase.
11	**18,699**	**+3,003**	Server has found and captured the waiting client thread.
12	19,365	+666	Server reversed the IPC connection. Client is now receiver.
13	20,135	+770	Server completed send of null reply.
14	20,364	+229	Server will disconnect current client, begin wait-receive processing for next client, and then will block.
15	**26,396**	**+6,032**	Client restarted and is resuming IPC.
16	26,572	+176	Client is about to exit the main IPC function and return to user mode having completed a complete round-trip null IPC.

Times reported in cycles.

server thread is finishing its previous IPC call. Also, when entering an IPC system call 700 cycles are required to initialize the timestamp buffer for recording; again, this overhead is only recorded in step 7 when the server thread starts a new IPC operation. (For the client, IPC system call entry overhead is recorded before step 1 while the exit overhead occurs after step 16.)

The five highlighted stages (stages 4, 7, 8, 11 and 15) are the obvious bottlenecks candidates. Before explaining the bottleneck candidates, the costs incurred by the other stages should be examined. First, consider the cost to get to stage 2 from stage 1—374 cycles. Even allowing 200 cycles for the timestamp code,[3] a lot of cycles were used just to get from the function entry point to the first timestamp before any "real work" gets done. In fact, there are only (in the Java source) two assignment statements and three conditionals (one while-loop test and two if statements) between the two stages. An explanation for this slowdown—the poor performance of the Kaffe compiler—will be discussed later, in Section 4.3.1.4.

The bottleneck candidate stages all involve finding a thread, "capturing" a thread or switching to a different thread. Threads can be "captured" by other threads in the kernel—the target thread is removed from its wait queue, implying that it will not be woken until the capturing thread explicitly wakes it or re-queues it. These parts of the Alta kernel contain the heaviest use of synchronization. Steps 7 and 15 include a thread switch and the best-case cost of a thread switch is 540 cycles (for a `wait/notify`-style thread switch). Of course, the middle of the IPC path is clearly not a best-case thread switch.

Additionally, by adding instrumentation code to the IPC path, the overall execution time of a null IPC was increased by roughly 7,000 cycles (3,200 of which are accounted for in stage 8), so in reality, the costs of each stage are somewhat smaller. Because the majority of the IPC code path is implemented in Java, improvements to the bytecode compiler would directly improve the performance of Alta's IPC.

[3]In a tight loop, the average cost of a timestamp is about 64 cycles; for a "cold-cache" test the cost is about 274 cycles.

The thread synchronization code in the bottleneck stages is a layer built, in Java, above the low-level synchronization primitives used within the virtual machine. By integrating the features required by the IPC code into the low-level synchronization primitives, the performance of the bottleneck stages should improve.

4.3.1.4 Code Generation

One result implied by the IPC code breakdown is that even simple operations are expensive in Alta. The code fragment presented in Figure 4.1 was used as a simple test case to compare the Kaffe dynamic bytecode compiler versus the `egcs` C compiler [14] and `gcj` [15], the Java front-end to `egcs`. The test function uses a common subset of C and Java syntax, and is similar in structure to the main loop in the IPC implementation. Table 4.7 compares the quality of the generated machine code. Notice that, even for such a small and simple fragment of code, the Kaffe JIT generates twice the number of instructions (taking four times as many bytes) and five times the number of memory references as either of the static, optimizing compilers, which implies there is significant room for improvement in converting bytecodes to native instructions with Kaffe. It is important to note that the `gcj` compiler used the bytecode generated by javac and not the Java source as input. The code generated by `gcj` and `egcs` is quite similar, despite the disparity in the inputs.

Table 4.7. A comparison of various Java native compilers.

Code Feature	**Java bytecodes**	**Kaffe JIT**	`egcs`	`gcj`
Input source	Java source	Class file	C source	Class file
Generated output	Class file	x86 code	x86 code	x86 code
Instructions Generated	31	65	28	26
Bytes used (NOP bytes)	55 (0)	227	68 (8)	56 (6)
Memory references	*N/A*	11	2	2
Stack references	11	12	2	2
Branch instructions	9	9	9	9

```
int simple(int ops, int rc)
{
        while (ops != 0)
        {
                if ((rc != 0) && (rc != 0x0F))
                {
                        return rc|0xFF00;
                }
                if ((ops & 0x100) != 0)
                {
                        ops = ops & ~0x100;
                }
                else if ((ops & 0x200) != 0)
                {
                        ops = ops & ~0x200;
                }
        }
        return 0;
}
```

Figure 4.1. A fragment of source code that is legal Java syntax and legal C syntax. This small piece of source code mimics the style of the main IPC loop in Alta.

Because `gcj` is not complete enough, as of this writing, to compile any useful portion of Alta's Java source, comparisons of the run-time differences for the compilers cannot be presented. Given the number of memory references made by the Kaffe-generated code, and the size of that code, it seems obvious that a significant gain in performance could be obtained by improving Alta's JIT compiler. Note that a JIT is not intended to rival the quality of code generated by an optimizing ahead-of-time compiler.

4.3.1.5 Process Creation

Alta can create a nested process in approximately $39ms$ (11.9 million cycles). The cost rises to $42ms$ if the cost of allocating and initializing the various objects required by the parent is included. The cost is measured as the timestamp difference from the parent process's call to `edu.utah.npm.core.Space.startMain()` to the entry to the child process's `main()` method. $39ms$ is an average of 11 trials which have a standard deviation of less than $1ms$. The time to start a child includes the time to fault in 26 classes including basic I/O classes, simple exceptions, and other miscellaneous classes.

In comparison, an Alta thread is created in about $0.2ms$ (see Section 4.3.1.1), and a `fork()` and `exec()` of a very simple C program under FreeBSD takes about $0.7ms$. Using `java.lang.Process.exec()` (based on FreeBSD's `fork()` and `exec()`) to start a new Kaffe process from within Kaffe requires approximately $300ms$. Because a large portion of the time spent starting a new Alta process is used to fault in the most basic classes, moving to a static definition of a process's typespace would probably improve the performance of process creation greatly (see Section 5.3). Additionally, unlike fork-and-exec in FreeBSD, which can simply share the executable's memory image between a parent and child, Alta re-links every critical class object in the context of the new process. Again, the cost of Alta's very dynamic class loading mechanism could be mitigated by using a more static definition of a process typespace. Lastly, the compiled code in Alta is process specific and thus this benchmark includes the time to compile and execute all of the methods invoked before `main()`—a number of constructors and `java.lang.ThreadGroup.add()`.

4.3.1.6 Interposition

To estimate the cost of interposition, two different scenarios are measured. First, the cost of interposing on the class name resolution of a child process is measured and second, the cost of interposing on the file system access interface is measured. Both of these tests use the Alta sandbox, a simple application that interposes on the parent interface, for class loading and memory control, and on the file system and file I/O interfaces.

First, to measure the cost of interposing on class name resolution, I executed, in the Alta sandbox, a simple program that dynamically loads a fixed number of classes. The loaded classes are minimal and require very little run-time initialization, so the time recorded for loading such a class should be dominated by the overhead of class loading. Running this test case as a child process of the sandbox measures the overhead of loading a class by faulting via IPC to a parent. These tests measure the worst case for loading a class because the class is not already loaded into the parent process. Table 4.8 presents the results. A parent process adds about 89,000 cycles (just under $0.3ms$ on a 300Mhz machine) of overhead to class loading. Because class loading is generally expensive anyway (most class files require more involved initialization than the minimal class files used in this test), this overhead does not seem unreasonable. Additionally, classes are only loaded once in a process. The performance of this operation could be improved by moving to an IPC implementation that takes advantage of idempotent semantics, or moving to a static definition of a process's class mapping.

Table 4.9 shows the overhead associated with interposing on the file read interface. A simple benchmark which makes 256 consecutive 128-byte reads from a file was repeated 34 times, and the resulting times averaged. The benchmark was run under zero to five sandbox processes. Interposing on the file read interface is approximately 10,000 cycles cheaper than interposing on the class loading interface. This difference is due to the more complex processing on both the client and server when handling class objects. The difference between copying a 128 byte buffer and sharing the buffer—several thousand cycles—is lost in the cost of the interposition.

Table 4.8. Class loading costs in nested processes.

Sandbox nesting depth	**Average time to load a class**	**Difference from previous**
0	206,000	-
1	291,000	85,000
2	370,000	79,000
3	460,000	90,000
4	547,000	87,000
5	653,000	106,000

Times reported in cycles.

Table 4.9. Interposed file read costs.

Sandbox nesting depth	Time to read 128 bytes (shared buffer)	Change	Time to read 128 bytes (buffer copy)	Change
0	2,800	-	2,900	-
1	66,000	63,200	73,900	71,000
2	128,700	62,700	144,300	70,400
3	196,400	67,700	222,600	78,300
4	266,000	69,600	297,300	74,700
5	337,500	71,500	387,800	90,500

Times reported in cycles.

4.3.1.7 Application performance

To test of the impact of interposition on actual application performance, Sun's Java compiler (a Java application) was run on Alta, under nested sandbox applications. The sandbox server interposes on the class loading interface, the file system interface and the file I/O interfaces. The sandbox does nothing but pass the interposed request on, so these results measure the minimum overhead for interposition. The compiler was given 24 different, small files to compile. Table 4.10 shows how long it took (in milliseconds) to compile the 24 files when the compiler

Table 4.10. Compilation costs in nested processes.

Sandbox nesting depth	Time to compile 24 files	Difference from previous
0	3,342	-
1	3,781	439
2	4,141	359
3	4,556	415
4	4,959	403
5	5,428	469

Times reported in milliseconds.

is nested within zero to five sandbox processes. The times reported are an average of 34 trials and do not include the cost of starting the virtual machine, only the time from the invocation of the compiler's entry point to the invocation of `java.lang.System.exit()` Note that JIT compilation of the compiler and some GC time are included.

Each additional level of nesting adds a fairly constant cost of $400ms$ to the compilation time. The most important aspect of this result is that the interposition costs in this table grow linearly with the number of interposition levels.

4.4 Comparison with Fluke

Comparing Alta to existing virtual machines provides a sense of the cost of adding processes to a Java virtual machine, but how does the system compare to existing, hardware-based operating systems? Because Alta is based on the OS model developed for the Fluke operating system, Fluke is the obvious choice for a comparison. This section begins with a comparison of how well the model from Fluke was able to map into Java, explains some of the differences and then presents some quantitive comparisons.

4.4.1 Implementation comparison

Obviously the interfaces and their semantics are borrowed from Fluke, but Alta also borrows structure from Fluke's implementation. For example, Alta's IPC implementation is, in large part, a Java translation of Fluke's IPC implementation (written in C). Many of the kernels internal structural details are similar. For example, Fluke's internal relationships between threads, ports, port sets, and IPC is maintained in Alta. Internally, the Alta kernel represents objects with the same structure as Fluke, but Alta takes advantage of Java's support for objects.

This structural similarity between the systems enables comparisons between the programming languages and their relative applicability to writing system software. The current state of Java code compilation, however, makes a comparison of the performance of Java versus C in system software untenable. The results in Section 4.3.1.4 demonstrate that there is potential for significant improvements in

Java compilation in the near term. Additionally, the Alta kernel is written in more of a C coding style than a highly object-oriented Java style, so kernel code should not suffer unduly from overhead due to dynamic dispatch—a traditional bottleneck in highly object-oriented systems [3].

While a lot of functionality was copied from Fluke, not all of Fluke's features were implemented in Alta. For example, Alta only implements reliable IPC and does not (yet) support the best-effort or at-least-once IPC mechanisms supported by Fluke. Originally, I thought reliable IPC would be sufficient, but there are two places where at-least-once IPC (for idempotent operations) would improve Alta. First, if the class fault used it, then class faults could be triggered in the middle of reliable IPC operations that cause a new class to be loaded. For example, when a process sends an object to another process that does not yet have the object's class loaded. Additionally, at-least-once IPC would use less memory and thus be a better candidate for delivering out-of-memory signals to a parent memory pool.

While Alta supports the Fluke condition variable and mutex objects, they are redundant with the native monitor support, via the `synchronized` keyword, in Java. Additionally, the kernel exceptions and per-thread exception handlers defined by the Fluke API are not required in Alta because of Java's native support for exception objects and exception handlers.

One important and compelling difference between Alta and Fluke is the treatment of objects that have both kernel-private and user-accessible portions. To protect kernel-private data, Fluke uses "schizophrenic" objects that have separate kernel and user-mode portions. In most cases the user-mode object is just a handle, and the kernel-half of an object is looked up from the handle (just as file descriptors work in traditional Unix systems). Alta, on the other hand, can take advantage of the language protections provided by Java and can make data members "private"—which hides them from untrusted user code without denying access to trusted kernel code associated with the object. Besides avoiding the cost of looking up a kernel object for each user-mode object, this flexible protection allows other optimizations. For example, the Alta kernel can charge the cost of allocating kernel state to the user at the time the user-mode portion is created—which simplifies the kernel implementation.

One final benefit of type safety is that user-visible kernel objects will always be correctly initialized before any operations are performed with them as the object's constructor is guaranteed to be invoked before the first usage. This constraint means code in the critical path does not need to check to make sure that an object is properly and completely initialized. Additionally, in Java, operations can be invoked only on legitimate objects, while in Fluke, a kernel operation can be invoked on any address and the kernel must verify the given address.

Another compelling difference between Alta and Fluke is the ability to safely share objects between processes in Alta. While the benefits of this have yet to be fully demonstrated, it seems reasonable to assume that being able to access data directly—without IPC or object serialization—and safely will improve performance and enable new applications. Keep in mind that sharing Java objects is not simply sharing of raw data; sharing includes the code to access and manipulate the object, too. Additionally, type safety can be leveraged to mitigate one of the traditional weaknesses of capability systems: capability propagation. In a traditional capability system, if one process hands a capability to another process, the original owner of the capability has effectively lost control over the propagation of that capability. The recipient of a capability can pass it on to other processes without informing the original owner. In a type-safe system, the recipient of a capability could be trusted code that is impenetrable to the process that receives the capability. Assuming that only trusted code can access the capability in a child process, then additional guarantees can be made. For example, the trusted code could guarantee that only legitimate operations are invoked on the capability, or that operations are only invoked if sufficient resources are available in the client.

The final distinction between Alta and Fluke is that Alta does not support state extraction for kernel objects, yet still provides an effective foundation for nested processes. As noted in section 3.2.7, Fluke kernel object state extraction is used for a comprehensive checkpointer, virtual memory servers, and process migration. The cost of this support, however, is a more complex kernel interface and implementation, specifically in the implementation of IPC and thread support [47]. Alta demonstrates that significant functionality—namely resource control—can be provided by the nested process model independent of support for total state extraction.

4.4.2 Performance comparison

The Fluke benchmark results were obtained on the same machine used for measuring Alta, a 300Mhz Intel Pentium II. Two versions of Fluke are measured. First, the August 1996 version, which contains the IPC path copied into Alta. The second version is the January 1999 version of Fluke, which includes a revamped IPC framework and a number of optimizations to support user-mode kernel objects [18]. Note that the new version of Fluke supports two modes of execution for in-kernel threads, a stack-per-thread (process model) or stack-per-processor (interrupt model) [21]. The interrupt model has a slight performance advantage, but the process model is the execution model used by Alta. Fluke was configured for process model execution when it was measured.

Table 4.11 compares Alta and the two versions of Fluke in terms of primitive OS operations. Alta outperforms the old version of Fluke in all of the listed operations, except for `Reference.compare()`. Alta loses on this function call because Alta's

Table 4.11. A comparison of Fluke and Alta.

Benchmark	Alta	Old Fluke	New Fluke
`Thread.self()`	13	378	4
Null system call	192	*N/A*	302
`Port.reference()`	891	1,363	649
`Reference.compare()`	2,911	1,361	6
Uncontested mutex lock/unlock	1,644	1,744	17
Thread Switch (`yield`)	185	*N/A*	519
Thread Switch (wait/notify)	540	2,866	1,268

Times reported in cycles.

`Reference` objects are correctly tracked by the object to which they point, and when a reference-able object is destroyed, all attached Reference objects are cleaned up. In contrast, Fluke's `References` can keep referenced objects "alive" even after those objects have been explicitly destroyed. This extra functionality requires an extra layer of locking in Alta—notice, though, that this extra locking does not unduly inflate the `Port.reference()` time. Additionally, `Reference.compare()` is implemented purely in Java and is thus hampered by the Kaffe JIT compiler.

When compared to the newest version of Fluke, Alta loses out to Fluke's user-mode code optimizations. All of the operations that Fluke completes faster than a null system call are, obviously, wholly user-mode operations. All of these optimizations could, however, be incorporated into Alta, as they all rely on the fact that Fluke exposes noncritical parts of kernel objects to user mode; a fact that holds true for all kernel objects in Alta. Still, Alta is more than twice as fast at thread switching, and has a cheaper null system call. (Note that an Alta system call is on par with a synchronized method invocation, which takes 156 cycles.) Overall, Alta demonstrates reasonable performance for fundamental operations when compared to Fluke.

For higher-level, more complex OS operations, Alta does not compare as well. Table 4.12 shows that for thread startup, process startup, and null IPC, Fluke outperforms Alta. Note that, in this table, the process start measurements are approximate, as they include some IPCs for transferring timestamps from the parent

Table 4.12. Comparison of OS operations in Alta and Fluke.

Benchmark	**Alta**	**Old Fluke**	**New Fluke**
Thread Start	55,413	7,806	3,794
Null IPC	18,524	7,519	3,843
Process Start	11m	1m	3m

Times are reported in cycles. For process start times, the "m" stands for "million."

process to the child process. Figure 4.2 shows that while Alta's IPC is consistently slower than IPC in either version of Fluke, the difference is constant. Figure 4.3 shows that the gap between Alta and Fluke is constant for all reasonable data sizes.

To get a better sense of why Alta IPC is three times slower than the old Fluke implementation's, Table 4.13 (p. 71) presents a breakdown similar to that presented in Table 4.6. This table includes a breakdown from the old Fluke IPC implementation. In this table, the differences between stages are comparable, except for stages 7, 8, 11, and 15, where Alta requires several thousand more cycles than Fluke to complete the stage. All four of these stages involve kernel-internal thread synchronization routines. In Alta this synchronization is written entirely in Java and duplicates functionality that is present in the virtual machine but which is not cleanly exposed to Java code; Alta could be modified to take advantage of this synchronization at a low level. Assuming this were the case, Alta would still be slower than Fluke because all the other stages of the IPC path are uniformly slower

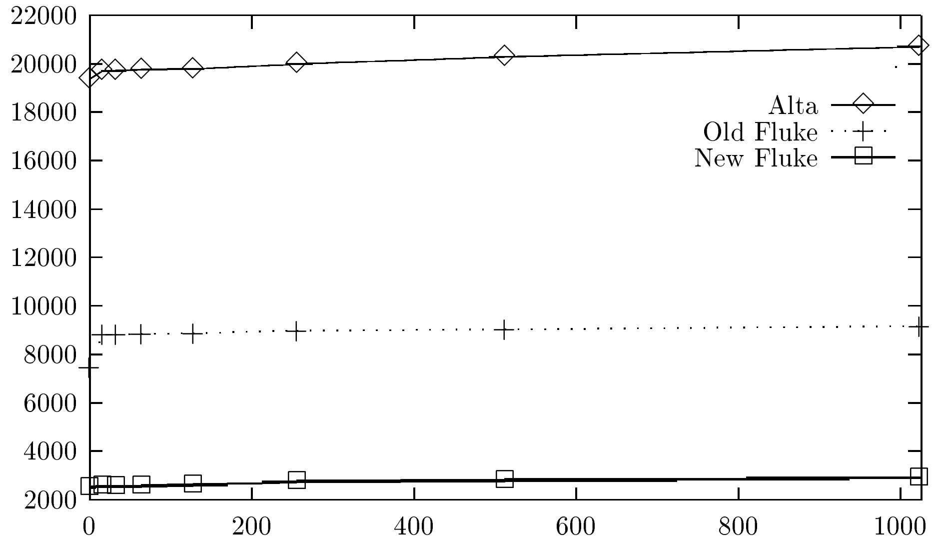

Figure 4.2. A comparison of IPC data transfer times for transfers between 0 and 1024 bytes.

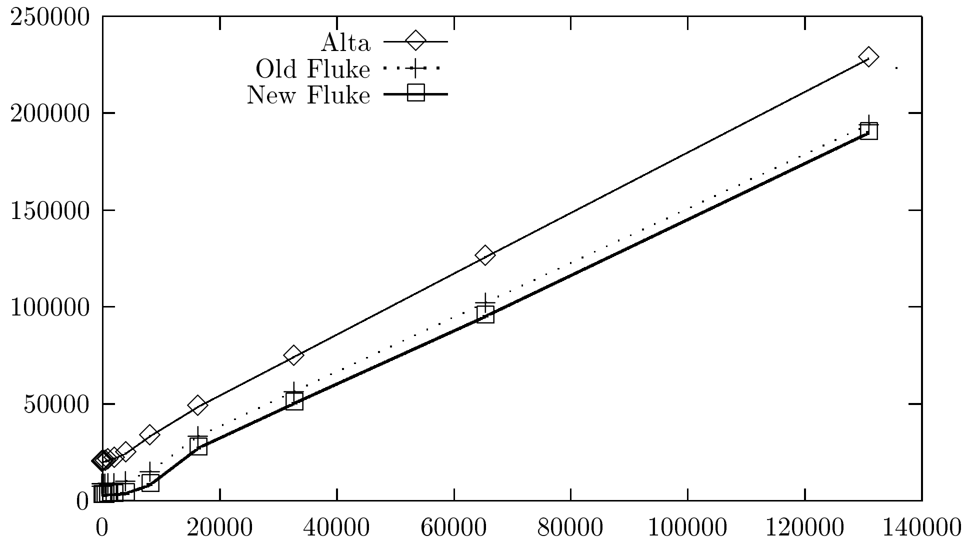

Figure 4.3. A comparison of IPC data transfer times for transfers between 0 and 128k bytes.

Table 4.13. Comparison of a null IPC cost breakdown between Fluke and Alta.

	Alta		Fluke		
	Time	**Diff.**	**Time**	**Diff.**	**Description**
1	0	0	0	0	Client entered main IPC loop.
2	374	+374	45	+45	Client will start "connect" phase.
3	796	+422	177	+132	Client cleared server connection.
4	2,558	+1,762	1,927	+1,750	Client found waiting server thread and captured it.
5	3,503	+945	2,116	+189	Client transferred its null request to the captured server thread.
6	3,798	+295	2,162	+46	Client will invert connection.
7	10,334	+6,536	3,381	+1,219	Server thread awoken, will return to user mode, and will immediately begin a new ack-send-(disconnect)-wait-receive call.
8	15,246	+4,912	4,086	+705	Server entered main IPC loop and will start an ack-send-(disconnect)-wait-receive. Includes measurement overhead.
9	15,453	+207	-	*N/A*	Server entered main IPC loop.
10	15,696	+243	4,142	+56	Server is about to begin handling the "ack-send" phase.
11	18,699	+3,003	4,789	+647	Server has found and captured the waiting client thread.
12	19,365	+666	-	*N/A*	Server reversed the IPC connection.
13	20,135	+770	4,966	+177	Server completed send of null reply.
14	20,364	+229	5,853	+887	Server will disconnect client then block.
15	26,396	+6,032	-	*N/A*	Client restarted and is resuming IPC.
16	26,572	+176	6,656	+803	Client is about to return to user mode.

Times are reported in cycles.

in Alta—ideally even this difference will diminish with improved Java compilation.

4.5 Conclusion

Alta demonstrates the feasibility of hosting operating system abstractions designed for an MMU-based architecture in a system based around the type safety provided by Java. Alta demonstrates that while there are some shortcomings to Java as a systems programming language, they can be worked around cleanly. Creating new processes in Alta is expensive, but most of this cost can be attributed to the high cost of dynamic class control. Using a static definition of a process's typespace would simultaneously improve loading times, by completely avoiding frequent and costly IPC calls, and increase the set of interprocess-shareable classes. While recent versions of Fluke strikingly outperform Alta on a number of benchmarks, extending the JIT compiler to support inlining should bring these benefits to Alta, too. While an in-depth comparison between the performance of these two systems is hampered by the quality of the native code generator in Kaffe, the initial results look promising for the performance of Java based systems.

CHAPTER 5

FUTURE WORK

There are many directions to pursue beyond this initial work. The most promising endeavor, in terms of novel research, would be to provide accounting and control over the resources used by the garbage collector, although this is addressed to some extent by KaffeOS (described in Section 6.1.2). The second major effort would be to provide a formal analysis of Alta's type system and prove type safety under this new system. These future directions and several others are explained in detail in this chapter.

5.1 Resource accounting and garbage collection

In a system that enforces total separation of processes, it is quite easy to "clean up" after terminating a process as there are no interdomain references to keep track of. The majority of resource accounting problems stem from fine-grained object sharing between domains with different resource principals. The current design for Alta provides processes with mechanisms to control interdomain sharing, but Alta only accounts for shared resources in a very simple manner.

The first weakness in Alta is exposed at object allocation time. The thread allocating the object is charged for the object's memory. This policy is generally good enough, but does not work if, for example, a server allocates an object and passes it to a client—there is no way to transfer the memory cost of the object in Alta. Alternatively, charging an object's memory to the processes that *use* the object might be more representative of the actual memory costs involved, but could lead to asynchronous memory exceptions. For example, consider a large object to which many domains hold a reference and where each domain is charged a portion of the total memory cost; if many of the domains release their reference to the object, the memory cost for the remaining domains will increase. Asynchronous exceptions resulting from such increases would make the already complicated task of writing multithreaded code far more complicated.

The second resource-related weakness is that Alta makes no attempt to charge particular processes for the resources required to run the garbage collector. This weakness makes Alta vulnerable to poorly-behaved or malicious processes, and weakens the system's ability to provide strong quality of service guarantees.

For example, if a process A has a small working set, a small memory limit, but a high allocation rate, it will necessarily invoke the global garbage collector frequently. Other processes in the system will be paused as the GC runs. Note that parent processes can constrain the number of invocations of the GC made by a process, but A should be able to collect its own heap without impacting other processes. One potential solution might be to restrict a process to an allocation *rate*, thus restricting the amount of garbage a process can potentially generate and indirectly constraining the work the garbage collector might need to do.

5.2 A formal analysis of the Alta type system

The simple statements defining Alta's type system in Section 3.2.2.1 would benefit from a formal analysis that proves the assertion that type safety in Alta cannot be compromised by colluding processes. The formal analysis of the current Java class loader system in [31] should provide terminology and a good basis for the proof.

5.3 Static class maps

There are several scenarios where the dynamic nature of Alta's IPC-based class name resolution provides unused flexibility. For example, if a parent process simply wants to deny access to all packages other than the standard `java.*` packages, or simply wants to remap a few particular classes to customized IPC-based classes, then the full set of class name mappings could be statically specified when the child process is created. This has the advantage of reducing the run time overhead for loading a new class. Additionally, a static class mapping scheme would provide the virtual machine with more information about how types map from one process into another. This information could be leveraged to make the rules on sharable types more flexible, specifically loosening the restriction on nonpolymorphic field types.

5.4 Reference revocation

The problems brought to resource accounting by interdomain references (see Section 3.2.5) could be alleviated by enhancing the virtual machine to support reference revocation [4]. For example, the Alta kernel could explicitly erase references to objects in a dying process, which would ensure that all of the objects owned by that process are successfully reclaimed. This would have the advantage, similar to a traditional hardware-based system, that the system could kill a process and all of its shared objects and be guaranteed that all of the objects would be reclaimed.

Supporting reference revocation would require solving a number of other hard problems [4]. First, finding all of the references to a revoked object would be difficult, especially in the face of C code running in the virtual machine and references on thread stacks. Additionally, the semantics of a thread executing in the context of a method on a revoked object are not clear, and the ramifications of revoking nontrivial objects such as threads, class loaders, and classes are also very complicated. Finally, the overhead of tracking object references (or finding all references on demand) would probably be great.

5.5 Integration of Fluke and Alta

Because Alta and Fluke share many of the same fundamental abstractions, combining the two systems into a mixed enforcement environment should be possible. In this mixed environment Java processes could intermingle with hardware-separated processes and could control each other's resources. For example, a Java-based web server could create a hardware-enforced subprocess in which untrusted binaries could be executed, or a native database system could create a Java-based subprocess that contains and manages client queries written in Java. The simplest approach would be to restrict interlanguage communication to a common, purely IPC-based protocol. A more ambitious project would be to unify the memory and CPU scheduling abstractions to make for more seemless interaction between Java and native code.

5.6 Alta applications

Another area of future work is to design and implement applications that take advantage of Alta's sharing model to provide unique and inexpensive services. For example, a file server in Alta could be implemented entirely in Java libraries. The shared file cache code would be executed entirely by clients; type safety would guarantee the integrity of the file cache and the file system. A system that truly exploits this sharing model would contain completely stateless servers that do no work and consume no resources "on their own"—all of the resource consumption is done by libraries in clients. The server, if any still exists, merely coordinates and maintains persistent information that outlasts individual clients.

One of the difficulties faced by such a system would be synchronization in the face of asynchronous process termination. The kernel is able to solve this problem by postponing asynchronous exceptions, but such a mechanisms is necessarily unavailable to arbitrary (i.e., untrusted) user mode servers.

CHAPTER 6

RELATED WORK

Alta touches on a large body of related work in two specific areas: java-based operating systems research and more general operating systems research.

6.1 Related Java work

There are currently six other Java-based projects that provide OS-style support for Java applications in a single virtual machine: the J-Kernel, KaffeOS, the Princeton JVM, Conversant, Sun's JavaOS, and the E-Communities capability system.

6.1.1 The J-Kernel

The J-Kernel [29] from Cornell is implemented entirely in Java and requires no changes or enhancements to the virtual machine. The J-Kernel concentrates on clean domain termination and provides this by enforcing complete separation of processes, even between domains that have some degree of trust between them. As pointed out by the J-Kernel authors, the "main problem with the approach taken in the J-Kernel is that data structures cannot be shared directly among tasks, thus not realizing one of the major potential gains of a language-based protection model" [30]. Additionally, the J-Kernel's cross-domain thread migration does not provide true separation, as a thread can block in another domain, out of reach of the owning domain. This model is sufficient for environments similar to applet environments, where the callee is trusted by the caller.

The J-Kernel uses on-the-fly LRMI stub generation to guarantee that *no* object references are passed between two domains. The stubs transparently convert any implicit cross-domain references into capabilities. The generated capabilities provide an indirect level of access to remote objects and insure memory separation for processes. Alta, on the other hand, allows two processes to share object references directly. Alta could use the J-Kernel's dynamic stub generator to transparently create IPC-based remote object interfaces for Java objects. Currently Alta's IPCIF interfaces are written by hand.

Because the J-Kernel has been restricted to an unmodified virtual machine, the support for control over memory and CPU usage is cumbersome or ineffective. CPU scheduling is done indirectly by manipulating thread priorities based on CPU usage. Memory accounting only takes into account memory directly used by Java-level objects; structures which are internal to the virtual machine are not charged to the owning process. The pure Java approach used to implement the J-Kernel allows it to take advantage of the best available Java virtual machine. Of course, resource management in the J-Kernel would easily benefit from providing direct virtual machine support as is done for Alta.

6.1.2 KaffeOS

KaffeOS is another Kaffe-based multiprocessing Java virtual machine from the University of Utah [6]. KaffeOS separates each Java process's heap and supports independent garbage collection of each heap. KaffeOS builds its notion of a process on top of Classloaders. Like Alta, KaffeOS will also support CPU inheritance scheduling for CPU management. KaffeOS has an even more restrictive form of interprocess sharing than Alta—the only mutable data that can be shared is primitive types. KaffeOS uses run-time write-barriers to prevent interheap references between processes. KaffeOS required no changes to the Java type system to support its sharing and communication mechanism.

Because KaffeOS is built on the same foundation as Alta, incorporating many of the engineering improvements—including garbage-collectible classes and CPU inheritance scheduling—from KaffeOS into Alta should be straightforward. On the other hand, reconciling the different process models and the different approaches to sharing might prove to be more difficult. For example, KaffeOS's per-process garbage collection relies on KaffeOS's restricted sharing model to avoid cross-process object references.

6.1.3 Princeton Multiprocessing JVM

Balfanz and Gong [7] describe a multiprocessing JVM developed to explore the security architecture ramifications of protecting applications from each other, as opposed to Sun's focus of only protecting the system from applications. They identify several areas of JDK version 1.2 that assume a single-application model, and propose extensions to allow multiple applications and to provide interapplication security. The focus of their multi-processing JVM is to explore the applicability of the JDK 1.2 security model to multiprocessing, and they rely on the existing, limited JDK infrastructure for resource control.

6.1.4 Conversant

A now defunct project from the Open Group Research Institute, Conversant [8] used a modified a Java virtual machine—running on Mach [1]—to support multiple Java processes. In Conversant, each Java process is provided with a separate, contiguous range of the virtual machine's address space, and a separate garbage collection thread. Conversant uses the extreme approach of total separation of processes within the virtual machine. Complete separation maximizes the effectiveness of resource accounting and control as all resource usage is trivially charged to the owning process. Conversant cannot, however, support any sharing between processes, and thus foregoes one of the major motivations for a type-safe-language based system.

6.1.5 JavaOS

Sun's JavaOS [43] was originally a complete platform written almost entirely in Java. It was described as a first-class OS for Java applications. From the scant literature available, JavaOS appears to provide a single Java virtual machine with no notion of a process or the ability to run multiple applications on that virtual machine. Sun Microsystems is replacing JavaOS with a new set of "Java operating systems": "JavaOS for Business," which only runs a single Java application; and “JavaOS for Consumers,” which is built on the Chorus microkernel OS [37] to provide the real-time guarantees needed in embedded systems. Both of these systems require a separate virtual machine for each Java application, which results in unnecessary overhead for duplication of virtual machine structures and initialization.

6.1.6 E-Communities capability system

Electronic Communities' E extends Java with a capability-based framework for distributed computing. E targets distributed Java systems (distributed across multiple physical hosts), but many of the specified capabilities would be useful on the local system. E's capabilities are static—they are granted at object load time—and E has no notion of a “process” or any other unit of resource control. E does provide significant infrastructure useful for distributed Java applications, such as digital signatures. Alta focuses on safety and resource control on the local node.

6.2 Related work in operating systems

Many research operating systems have used software protection mechanisms. Some, such as Pilot, use software-based protection throughout the system. Others, such as SPIN, use software-based protection only for kernel extensions.

6.2.1 Fluke

Fluke is a microkernel that only uses hardware protection mechanisms for process separation. The nested process model was first implemented in Fluke and much of Fluke's implementation served as a blueprint for Alta's implementation. Fluke was described in detail in section 2.2 and compared to Alta in section 4.4.

6.2.2 Language-based operating systems

6.2.2.1 Pilot and Cedar

Pilot [36] and Cedar [44] are two of the earliest language-based operating systems. The whole of these systems (including user tools, OS internals, etc) was written in the Cedar language. All processes operated in a single address space. Because Cedar and Pilot were designed for single-user workstations, isolation of malicious or erroneous processes was not a goal and remained unenforceable. While the Cedar language provided many safety properties, the system did not prevent resource hoarding or the corruption of system data by malicious processes. Additionally, the language supported a set of “unsafe” features that were used in the lowest levels of the system to circumvent many of the language’s safety properties. In a Java system these “unsafe” features are written in C and are unavailable in the language itself.

6.2.2.2 Oberon and Juice

The Oberon language and operating system [53] shares many of Java’s features, although it is a nonpreemptive, single-threaded system. The designers assumed that all operations undertaken on Oberon would be inherently short and they assumed any compute-intensive work would be offloaded to a compute server. These assumptions were valid when this system was designed in the mid-1980s. Background tasks such as the garbage collector are implemented as consecutive calls to procedures, where “interruption” can only occur between top-level procedure calls (called *commands* in Oberon). All state for these background tasks is global. While protection between tasks is enforced by the language, the exposure of global state to all top-level procedure calls and the uninterruptibility of commands mean that isolation of tasks is not enforceable.

The Juice project [24] uses Oberon and Slim Binaries [25] to provide a Java-like environment for downloaded code. Juice is the conceptual equivalent of a Java virtual machine, except that it does not interpret bytecodes—Juice expects binaries in the Slim Binary format, which are converted to native code and executed directly. Juice improves the distribution of platform independent binaries in two ways. First, by distributing a form closer to a compiler-generated parse tree, Juice binaries, unlike Java bytecodes, do not need to be verified. Second, Juice binaries provide a more compact representation of a program than Java class files. Unlike Alta, Juice does not address resource consumption or accounting.

6.2.2.3 Inferno

Inferno [17] is a system developed at Bell Laboratories. Inferno consists of the Limbo programming language and the Dis virtual machine. Limbo and Dis provide type safety and verification facilities similar to Java's. Inferno is oriented to network applications and provides data streaming and strong networking support. Inferno uses an extended file system namespace to provide applications with resources (similar to Plan 9 [35]) and uses namespace controls to restrict application access to those resources. Inferno does not support memory controls and does not appear to maintain a clear distinction between application and kernel.

6.2.2.4 NewtonOS

NewtonOS [50] is an operating system that supports communication and synchronization between processes within a single address space. Applications are written in the *NewtonScript* scripting language. Because *NewtonScript* supports pointers, the CPU's paging hardware must be used to provide protection between processes. The system, however, was only designed to run highly cooperative processes that are mutually trusting. Under these assumptions interprocess protections are not strictly necessary.

6.2.2.5 SPIN

SPIN [9] is an extensible operating system kernel that lets users load and link extensions, written in Modula-3, into the kernel. These extensions can interact tightly with the kernel or other kernel extensions to perform virtual memory operations, application-specific disk access, or other system-level functions. Unlike Cedar, SPIN provides sufficient protection to allow untrusted modules but has no strict resource controls. The language, with assistance from the linker, keeps extensions from interfering with each other and from interfering with critical parts of the kernel. To maintain control over extensions SPIN adds features to the language, such as `ephemeral` procedures—in this case, to maintain preemptibility of an extension.

SPIN uses software-based protection only for kernel extensions; regular processes are managed with traditional hardware-based techniques. SPIN was designed and optimized to provide a very rich interface to kernel extensions and does not provide a general operating system interface for user mode processes. There is no notion of "process" within the kernel and thus no interprocess communication or process management. SPIN does not limit the memory resources of kernel extensions and the only CPU management supported is termination of "ephemeral" extensions that have exhausted their CPU allocation.

6.2.2.6 Vino

VINO is a software-based (but not language-based) extensible system [38] that addresses some of the resource control issues raised by SPIN by wrapping kernel extensions within transactions. When an extension exceeds its resource limits, it can be safely aborted (even if it holds kernel locks), and its resources can be recovered. Extensions, and the resources used by an extension, are shared equally among all users; extensions do not run under the identity of the user code they are executing on behalf of.

Like SPIN, VINO only uses software-based protection for managing kernel extensions. Regular user-mode processes are managed with traditional hardware-based page protections.

CHAPTER 7

CONCLUSION

Alta demonstrates the applicability of operating system concepts to a Java virtual machine, leveraging type-safety where a traditional operating system would leverage hardware-based protection mechanisms. Alta is also evidence that standard operating system abstractions can be easily implemented in Java, and that traditional kernel implementation techniques are applicable in a Java-based operating system.

The work presented in this thesis makes the following four additional contributions to language-based operating system research:

- *Extensions to the Java type system to support safe sharing between different applications and a facility for renaming classes:* Definitions of class and class equality in a system with partially inconsistent typespaces were presented along with a simple algorithm for determining if two classes are safe for sharing between disjoint typespaces.

- *Implementation of the Fluke nested process model in Java:* Alta demonstrates the applicability of Fluke's nested process model to Java. By duplicating the design, and in many ways the implementation, of Fluke in Alta, I was able to evaluate the influence of Java's programming model on Alta's structure. Additionally, using several detailed performance studies, I identified some of the strengths and weaknesses of Java as a systems programming language.

- *Support for multiple applications in a single Java virtual machine:* Alta demonstrates that a language-based operating system can, to a large degree, control and separate applications as effectively as a traditional, hardware-based operating system without sacrificing Java's safe, fine-grained sharing.

- *Mechanisms for resource control in a Java virtual machine:* Direct mechanisms to account for the total memory usage of a Java application were demonstrated for Alta. Capabilities, a standard operating system abstraction, were demonstrated as a mechanism for controlling secondary resources such as files and the file system.

APPENDIX A

LOW-LEVEL NESTED PROCESS MODEL API IN ALTA

The public interfaces for the following classes, which comprise the Alta kernel interface, are listed in this appendix. For each class, any public fields, constructors and methods are listed.

Classes

```
edu.utah.npm.core.ClassHandle
edu.utah.npm.core.Cond
edu.utah.npm.core.Debug
edu.utah.npm.core.IPC
edu.utah.npm.core.IPCPayload
edu.utah.npm.core.IPCWaitReceiveReturn
edu.utah.npm.core.MemPool
edu.utah.npm.core.Mutex
edu.utah.npm.core.Port
edu.utah.npm.core.PortSet
edu.utah.npm.core.Reference
edu.utah.npm.core.Space
edu.utah.npm.core.Thread
```

final class `edu.utah.npm.core.ClassHandle`**:**

Constructors:

ClassHandle()

Methods:

final void ClassHandle()

final void ClassHandle(**Class** c)
final void useClass(**Class** cl)
final void disallow()
final boolean useWhatIUse(**String** name)

final class `edu.utah.npm.core.Cond`:

Extends Refable

Constructors:

Cond()
Cond(**int** hashCode)

Methods:

void wait(**Mutex** mutex)
void signal()
void broadcast()
void destroy()
String toString()
void finalize()

final class `edu.utah.npm.core.Debug`:

Constructors:

Debug()

Methods:

static void assertCheck(**boolean** expr, **String** exprStr, **String** file, **int** line)
static void assertCheck(Object ref, **String** exprStr, **String** file, **int** line)
static void dumpStats()
static void println(**String** str)
static void print(**String** str)
static void printInt(**int** i)
static void printIntHex(**int** i)
static void printIPCOps(byte ops)
static void printCharArray(char[] array, **int** start, **int** end)
static void exit()

static void threadInfo(Thread th)
static void printStackTrace()
static boolean doTrace(**int** flag)
static void otsan()
static void otsan(**String** str)
static void notImplemented()
static void notImplemented(**String** expl)
static void dPanic(**String** str)

final class `edu.utah.npm.core.IPC`:

Fields:

static final int RC_OK;
static final int RC_CONNECT_INVALIDDEST;
static final int RC_ACK_DISCONNECTED;
static final int RC_SEND_DISCONNECTED;
static final int RC_OVER_DISCONNECTED;
static final int RC_RECV_DISCONNECTED;
static final int RC_RECV_MOREDATA;
static final int RC_RECV_MOREREFS;
static final int RC_RECV_MOREOBJS;
static final int RC_RECV_REVERSED;

Methods:

static final int clientConnectSend(**Reference** destPortRef, **IPCPayload** payload)
static final int clientAckSend(**IPCPayload** payload)
static final int clientSend(**IPCPayload** payload)
static final int clientConnectSendOverReceive(**Reference** destPortRef, **IPCPayload** payload)
static final int clientAckSendOverReceive(**IPCPayload** payload)
static final int clientSendOverReceive(**IPCPayload** payload)
static final int clientOverReceive(**IPCPayload** payload)
static final int clientReceive(**IPCPayload** payload)
static final int clientDisconnect()
static final int clientAlert()
static final int serverAckSend(**IPCPayload** payload)
static final int serverSend(**IPCPayload** payload)

static final void serverAckSendWaitReceive(**IPCPayload** payload, **IPCWaitReceiveReturn** out_rc)
static final int serverAckSendOverReceive(**IPCPayload** payload)
static final void serverSetupWaitReceive(**IPCPayload** payload, **PortSet** pset, **IPCWaitReceiveReturn** out_rc)
static final void serverWaitReceive(**IPCPayload** payload, **IPCWaitReceiveReturn** out_rc)
static final void serverSendWaitReceive(**IPCPayload** payload, **IPCWaitReceiveReturn** out_rc)
static final int serverSendOverReceive(**IPCPayload** payload)
static final int serverOverReceive(**IPCPayload** payload)
static final int serverReceive(**IPCPayload** payload)
static final int serverDisconnect()
static final int serverAlert()
static final String errorToString(**int** rc)

class `edu.utah.npm.core.IPCPayload`:

Fields:

byte[] sendBuf;
int sendBufStart;
int sendBufEnd;
byte[] recvBuf;
int recvBufStart;
int recvBufEnd;
Reference[] sendRefTab;
int sendRefTabStart;
int sendRefTabEnd;
Reference[] recvRefTab;
int recvRefTabStart;
int recvRefTabEnd;
Object[] sendObjTab;
int sendObjTabStart;
int sendObjTabEnd;
Object[] recvObjTab;
int recvObjTabStart;
int recvObjTabEnd;

Constructors:

IPCPayload()

class `edu.utah.npm.core.IPCWaitReceiveReturn`:

Fields:

int rc;
Object alias;

Constructors:

IPCWaitReceiveReturn()

final class `edu.utah.npm.core.MemPool`:

Extends `edu.utah.npm.core.Refable`

Constructors:

MemPool(**long** size, **Reference** pPoolRef, **Reference** spaceRef, **Reference** keeperPortRef)
MemPool(**long** size, **Reference** pPoolRef, **Reference** spaceRef, **Reference** keeperPortRef, **int** hashCode)

Methods:

long getSize()
long getAvailable()
void destroy()
String toString()

final class `edu.utah.npm.core.Mutex`:

Extends `edu.utah.npm.core.Refable`

Constructors:

Mutex()
Mutex(**int** hashCode)

Methods:

void destroy()
void lock()
void unlock()
boolean trylock()
String toString()

class `edu.utah.npm.core.Port`:

Extends `edu.utah.npm.core.Refable`

Constructors:

Port(**Object** alias, **Reference** portSet)
Port(**Object** alias, **Reference** portSet, **int** hashCode)

Methods:

void setAlias(**Object** alias)
void destroy()
String toString()

final class `edu.utah.npm.core.PortSet`:

Extends `edu.utah.npm.core.Refable`

Constructors:

PortSet()
PortSet(**int** hashCode)

Methods:

void destroy()
String toString()

final class `edu.utah.npm.core.Reference`**:**

Extends `edu.utah.npm.core.NPMObject`

Fields:

static final Reference NullReference;

Constructors:

Reference()
Reference(**Refable** refObj)
Reference(**int** hashCode)
Reference(**Refable** refObj, **int** hashCode)

Methods:

Class type()
void destroy()
boolean compare(**Reference** otherRef)
boolean equals(**Object** obj)
void nullify()
boolean check()
String toString()

final class `edu.utah.npm.core.Space`**:**

Extends `edu.utah.npm.core.Refable`

Constructors:

Space(**Reference** keeperPort, **Reference** memPool)
Space(**Reference** keeperPort, **Reference** memPool, **int** hashCode)

Methods:

static Reference currentTaskKeeperPort(**Reference** out_ref)
static Reference currentTaskMemPool(**Reference** out_ref)
void startMain(**String** threadName, **String** initialObj, **String**[] initialArgs)
void destroy()
String toString()

class `edu.utah.npm.core.Thread`:

Extends `edu.utah.npm.core.Refable`

Constructors:

Thread(**Reference** spaceRef, **Reference** schedPortRef, **Reference** clientRef, **Reference** serverRef)
Thread(**Reference** spaceRef, **Reference** schedPortRef, **Reference** clientRef, **Reference** serverRef, **int** hashCode)

Methods:

static final void nullsyscall()
static final Thread self()
static final void interrupt(Thread thread)
static final boolean interrupted(Thread thread)
final boolean getClient(**Reference** out_clientThreadRef)
final void setClient(**Reference** clientThreadRef, **boolean** isSender)
final boolean getServer(**Reference** out_serverThreadRef)
final void setServer(**Reference** serverThreadRef, **boolean** isSender)
final void destroy()
static final void cancel(Thread th)
String toString()
String debugInfo()

class `edu.utah.npm.core.Exception`:

Extends `java.lang.Exception`

Constructors:

Exception()
Exception(**String** message)

class `edu.utah.npm.core.Cancelation`:

Extends `java.lang.Error`

Constructors:

Cancelation()
Cancelation(**String** message)

class `edu.utah.npm.core.DestroyedObjectError`:

Extends `edu.utah.npm.core.InvalidObjectError`

Constructors:

DestroyedObjectError()
DestroyedObjectError(**String** detail)

class `edu.utah.npm.core.Error`:

Extends `java.lang.Error`

Constructors:

Error()
Error(**String** message)

class `edu.utah.npm.core.Insanity`:

Extends `java.lang.Error`

Constructors:

Insanity()
Insanity(**String** message)

class `edu.utah.npm.core.InvalidObjectError`:

Extends `edu.utah.npm.core.Insanity`

Constructors:

InvalidObjectError()
InvalidObjectError(**String** detail)

APPENDIX B

IPCIF: ALTA IPC-BASED INTERFACES

The set of IPCIF interfaces defined for Alta are described here. These interfaces are designed to support the basic responsibilities of a parent process and to support the standard Java class libraries, as of JDK 1.0.2, but without support for an AWT. These interfaces are analogous to the "Common Protocol" interfaces in Fluke.

B.1 Parent interface

The parent interface is the "bootstrap" interface from which a process gains access to all other IPCIF services. Every `edu.utah.npm.core.Space` object has a port reference associated with it. Every process that creates a subprocess must also handle the parent interface requests for that process.

`ClassForName(String name)` Returns `ClassHandle`

Given a fully qualified class name, return a `edu.utah.npm.core.ClassHandle` object representing the class to bind that name to in the child.

`LastThreadDied()` Returns `boolean`

The child process will invoke this method when the last thread in the process has expired.

`Exit()` Returns `boolean`

Perform an explicit process exit.

`GetStandardIn()` Returns `Reference`

The child will invoke this method as it is being initialized to setup its default input stream. The `edu.utah.npm.core.Reference` returned should be compatible with the file interface.

`GetStandardOut()` Returns `Reference`

The child will invoke this method as it is being initialized to setup its default output stream. The `edu.utah.npm.core.Reference` returned should be compatible with the file interface.

`GetStandardErr()` Returns `Reference`

The child will invoke this method as it is being initialized to setup its default error stream. The `edu.utah.npm.core.Reference` returned should be compatible with the file interface.

`ServiceMapServerReq()` Returns `Reference`

The child will invoke this method to get a handle on the "Service Mapper" service (see Section B.2). The service mapper is a basic name service, mapping string names to ports. The returned `edu.utah.npm.core.Reference` is the port through which all other services are acquired.

B.2 Service map interface

The Service Map is a generic name to port reference binding service. It supports only one method.

`handlePortForName(String name)` Returns `Reference`

This method is invoked with a `java.lang.String` argument. The associated port reference is returned.

B.3 MemPool interface

The MemPool Interface is used by parent processes that manage the memory resources of a nested child. The child's interface to the garbage collector (invocation of the garbage collector and invocation of the finalizer) are accomplished through this interface.

`OutOfMemory(int shortAmt)` Returns `boolean`

Invoked directly by the VM, this method indicates that a process has insuffiicent memory in its MemPool (requiring `shortAmt` to proceed).

`InvokeGC()` Returns `void`

Invoke the GC.

`InvokeFinalizer()` Returns `void`

Invoke the finalizer thread.

`AvailableMemory()` Returns `long`

Returns the amount of memory available in child's MemPool.

`TotalMemory()` Returns `long`

Returns the total amount of memory in the child's MemPool.

B.4 File system interface

The File System Interface supports method for performing file-system level operations such as opening a file, renaming files, etc. These methods are used by the IPC-based version of the `java.io.File` class.

`CanRead(String path)` Returns `boolean`
Returns true if the file specified by `path` is readable by the current process.

`CanWrite(String path)` Returns `boolean`
Returns true if the file specified by `path` is writeable by the current process.

`Exists(String path)` Returns `boolean`
Returns true if the file specified by `path` exists.

`IsDirectory(String path)` Returns `boolean`
Returns true if the file specified by `path` is a directory.

`IsFile(String path)` Returns `boolean`
Returns true if the file specified by `path` is a regular file.

`LastModified(String path)` Returns `long`
Returns the last modification timestamp (in milliseconds) of the file specified by `path`.

`FileSize(String path)` Returns `long`
Returns the size (in bytes) of the file specified by `path`.

`Delete(String path)` Returns `boolean`
Deletes the file specified by `path`; returns true for success and false for failure.

`Rename(String from, String to)` Returns `boolean`
Renames the file specified by `from` to `to`; returns true for success and false for failure.

`Mkdir(String path)` Returns `boolean`
Creates the directory specified by `path`; returns true for success and false for failure. Does not create any required parent directories.

`DirList(String path)` Returns `String[]`
Returns an array of `java.lang.String` objects, one for each entry in the directory given by `path`.

`Open_read(String path)` Returns `Reference`
Returns a port reference to a File I/O Interface, corresponding to the file specified by `path`. The file is opened in read-only mode.

`Open_write(String path)` Returns `Reference`
Returns a port reference to a File I/O Interface, corresponding to the file specified by `path`. The file is opened for reading.

`Open_read_write(String path)` Returns `Reference`
Returns a port reference to a File I/O Interface, corresponding to the file specified by `path`. The file is opened in read-write mode.

B.5 File I/O interface

The File I/O Interface represents a single open file.

`Read(byte[] buffer, int toRead)` Returns `int`
Reads `toRead` bytes into `buffer` from this file. Returns the actual number of bytes read.

`Write(byte[] buffer)` Returns `int`
Write the entire contents of `buffer` into this file. Returns the actual number of bytes written.

`Skip(long bytes)` Returns `long`
Skip forward `bytes` bytes in this file.

`Ftell()` Returns `long`
Return the current offset in this file.

`Fseek(long offset, int whence)` Returns `long`
Move the current offset pointer `offset` bytes in the `whence` direction (from the beginning, from the end, or from the current position).

`Close()` Returns `void`
Close this file.

B.6 Network I/O interface

The Network I/O Interface corresponds to the File System Interface, but, obviously, provides an interface to networking services.

`Socket(boolean streamsocket)` Returns `Reference`
Create a stream socket or datagram socket (depending on `streamsocket`). Returns a port reference to a File I/O interface.

`Bind(byte[] addr, int port)` Returns `void`
Bind the current descriptor to the given IP address and port.

`Listen(int count)` Returns `void`

Flag the current descriptor to accept connections and to buffer count of them.

`Accept(byte[] resultAddr, Reference acceptPortRef)` Returns `int`
Accept a connection on the current descriptor. The resultAddr is the address of the remote connection and acceptPortRef will point to the new File I/O descriptor representing the connection.

`Connect(byte[] address, int remotePort)` Returns `int`
Connection to the given address and remotePort using the current descriptor.

`DGSocket()` Returns `Reference`
Create a datagram socket from the current descriptor.

`Recv()` Returns `int`
Receive a buffer of data on the current datagram descriptor.

`Send()` Returns `int`
Send a buffer of data on the current datagram descriptor.

`Close()` Returns `void`
Close the current socket.

REFERENCES

[1] Accetta, M. J., Baron, R. V., Bolosky, W. J., Golub, D. B., Rashid, R. F., Tevanian, A., and Young, M. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference* (June 1986), pp. 93–112.

[2] Adl-Tabatabai, A.-R., Langdale, G., Lucco, S., and Wahbe, R. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, May 1996), pp. 127–136.

[3] Aigner, G., and Hölzle, U. Eliminating virtual function calls in C++ programs. In *ECOOP '96: 10th European Conference on Object-Oriented Programming Proceedings* (July 1996), Lecture Notes in Computer Science 1098, Springer Verlag, pp. 142–166. ISBN: 3-540-61439-7.

[4] Back, G. V. Safe memory revocation in Java. Unpublished proposal, June 1998.

[5] Back, G. V., and Hsieh, W. C. Drawing the red line in Java. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems* (Rio Rico, Arizona, Mar. 1999).

[6] Back, G. V., Tullmann, P. A., Stoller, L. B., Hsieh, W. C., and Lepreau, J. Java operating systems: Design and implementation. Tech. Rep. 98-015, Department of Computer Science, University of Utah, Aug. 1998.

[7] Balfanz, D., and Gong, L. Experience with secure multi-processing in Java. In *Proceedings of the Eighteenth International Conference on Distributed Computing Systems* (May 1998). Available electronically at `http://-www.cs.princeton.edu/sip/pub/icdcs.html`.

[8] Bernadat, P., Feeney, L., Lambright, D., and Travostino, F. Java sandboxes meet service guarantees: Secure partitioning of CPU and memory. Tech. Rep. TOGRI-TR9805, The Open Group Research Institute, June 1998.

[9] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, Dec. 1995), pp. 267–284.

[10] Bracha, G., and Cook, W. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications* (Ottawa, Canada, Oct. 1990), ACM Press, pp. 303–311.

[11] Chan, P., and Lee, R. *The Java Class Libraries: Volume 2*, second ed. The Java Series. Addison-Wesley Publishing Company, Nov. 1997.

[12] Chan, P., Lee, R., and Kramer, D. *The Java Class Libraries: Volume 1*, second ed. The Java Series. Addison-Wesley Publishing Company, Nov. 1997.

[13] Clements, G., and Morrison, G. D. Kore—an implementation of the Java core class libraries. Available electronically at `ftp://sensei.co.uk/-misc/kore.tar.gz` or `http://www.cs.utah.edu/flux/java/kore/`.

[14] Cygnus Solutions. egcs project home page, 1998. Available electronically at `http://egcs.cygnus.com/`.

[15] Cygnus Solutions. The GCJ homepage, 1998. Available electronically at `http://sourceware.cygnus.com/java/gcj.html`.

[16] Dean, D., Felten, E. W., and Wallach, D. S. Java security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy* (Oakland, California, May 1996).

[17] Dorward, S., Pike, R., Presotto, D. L., Ritchie, D., Trickey, H., and Winterbottom, P. Inferno. In *Proceedings of the 42nd IEEE Computer Society International Conference* (San Jose, CA, Feb. 1997), pp. 241–244.

[18] Ford, B. A. User-memory-based kernel objects. University of Utah. Postscript slides available from `http://www.cs.utah.edu/flux/fluke/`, 1995.

[19] Ford, B. A., Back, G. V., Benson, G., Lepreau, J., Lin, A., and Shivers, O. The Flux OSKit: A substrate for OS and language research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (St. Malo, France, Oct. 1997), pp. 38–51.

[20] Ford, B. A., Hibler, M. J., and Flux Project Members. Fluke: Flexible μ-kernel Environment (draft documents). University of Utah. Postscript and HTML available from `http://www.cs.utah.edu/flux/fluke/`, 1996.

[21] Ford, B. A., Hibler, M. J., Lepreau, J., McGrath, R., and Tullmann, P. A. Interface and execution models in the Fluke kernel. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, Feb. 1999), USENIX Association.

[22] Ford, B. A., Hibler, M. J., Lepreau, J., Tullmann, P. A., Back, G. V., and Clawson, S. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, Washington, Oct. 1996), USENIX Association, pp. 137–151.

[23] Ford, B. A., and Susarla, S. CPU inheritance scheduling. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, Washington, Oct. 1996), USENIX Association, pp. 91–105.

[24] Franz, M., and Kistler, T. Introducing Juice. Available electronically at `http://www.ics.uci.edu/~juice/intro.html`, Oct. 1996.

[25] Franz, M., and Kistler, T. Slim binaries. *Communications of the ACM 40*, 12 (Dec. 1997), 87–94.

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

[27] Gosling, J., and McGilton, H. The Java language environment: A white paper. Tech. rep., Sun Microsystems Computer Company, 1996. Available electronically at `http://java.sun.com/doc/language_environment/`.

[28] Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D., and von Eicken, T. SLK: A capability system based on safe language technology. Available electronically at `http://www2.cs.cornell.edu/SLK/papers.html`, 1997.

[29] Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D., and von Eicken, T. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference* (New Orleans, Louisiana, 1998), pp. 259–270.

[30] Hawblitzel, C., and von Eicken, T. A case for language-based protection. Tech. Rep. 98-1670, Department of Computer Science, Cornell University, Mar. 1998.

[31] Liang, S., and Bracha, G. Dynamic class loading in the Java virtual machine. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (Vancouver, British Columbia, Canada, Oct. 1998).

[32] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley Publishing Company, Jan. 1997.

[33] McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.4BSD Unix Operating System.* Addison-Wesley UNIX and Open Systems Series. Addison-Wesley Publishing Company, 1996. ISBN: 0-201-54979-4.

[34] Necula, G. C., and Lee, P. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, Washington, Oct. 1996), USENIX Association, pp. 229–243.

[35] Presotto, D. L., Pike, R., Thompson, K., and Trickey, H. Plan 9, a distributed system. In *Proceedings of the Spring 1991 EurOpen Conference* (Tromsø, Norway, May 1991), EurOpen, pp. 43–50.

[36] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An operating system for a personal computer. *Communications of the ACM 23*, 2 (1980), 81–92.

[37] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Léonard, P., and Neuhauser, W. CHORUS distributed operating system. *Computing Systems 1*, 4 (Fall 1988), 305–370.

[38] Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, Washington, Oct. 1996), USENIX Association, pp. 213–227.

[39] Sirer, E. G., Gregory, A. J., McDirmid, S., and Bershad, B. N. The Kimera project. Available electronically at `http://kimera.-cs.washington.edu/`.

[40] Sirer, E. G., Gregory, A. J., McDirmid, S., and Bershad, B. N. The Kimera project: Flaws in Java implementations. Available electronically at `http://kimera.cs.washington.edu/flaws/`.

[41] Stallman, R. M. *GNU Emacs Manual*, thirteenth ed. Free Software Foundation, 1997.

[42] Sun Microsystems. The HotJava(tm) product family. Available electronically at `http://java.sun.com/products/hotjava/index.html`.

[43] Sun Microsystems, Inc. JavaOS: The standalone Java application platform, Feb. 1997. Available electronically at `http://www.javasoft.com/-products/javaos/`.

[44] Swinehart, D. C., Zellweger, P. T., Beach, R. J., and Hagmann, R. B. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems 8*, 4 (Oct. 1986), 419–490.

[45] Tullmann, P. A., and Lepreau, J. Nested Java processes: OS structure for mobile code. In *Proceedings of the Eighth ACM SIGOPS European Workshop* (Sintra, Portugal, Sept. 1998), ACM, pp. 111–117.

[46] Tullmann, P. A., Lepreau, J., Ford, B. A., and Hibler, M. J. User-level checkpointing through exportable kernel state. In *Proceedings of the Fifth International Workshop on Object Orientation in Operating Systems* (Seattle, Washington, Oct. 1996), IEEE, pp. 85–88.

[47] TULLMANN, P. A., TURNER, J., MCCORQUODALE, J. D., LEPREAU, J., CHITTURI, A., AND BACK, G. Formal methods: A practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems* (Cape Cod, Massachusetts, May 1997), pp. 20–25.

[48] VOLANO. Volano and JavaWorld release first java virtual machine server-side scalability report, July 1998. Available electronically at `http://www.-volano.com/vn072098.html`.

[49] WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. Extensible security architectures for Java. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Oct. 1997), pp. 116–128.

[50] WELLAND, R., SEITZ, G., WANG, L.-W., DYER, L., HARRINGTON, T., AND CULBERT, D. The Newton operating system. In *Proceedings of the 1994 IEEE Computer Conference* (San Francisco, California, 1994), IEEE.

[51] WILKINSON, T. J. Kaffe—a virtual machine to compile and interpret Java bytecodes. Available electronically at `http://www.transvirtual.com/-kaffe.html` and `http://www.kaffe.org/`.

[52] WIRTH, N. The programming language Oberon. *Software—Practice and Experience 18*, 7 (1988), 671–690.

[53] WIRTH, N., AND GUTKNECHT, J. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley Publishing Company, 1992.

[54] ZDNET. Your guide to Java for 1998, 1998. Available electronically at `http://www.zdnet.com/pcmag/features/java98/index.html`.