The KaffeOS Java Runtime System

GODMAR BACK
Virginia Polytechnic Institute and State University
and
WILSON C. HSIEH
Google, Inc.

Single-language runtime systems, in the form of Java virtual machines, are widely deployed platforms for executing untrusted mobile code. These runtimes provide some of the features that operating systems provide: interapplication memory protection and basic system services. They do not, however, provide the ability to isolate applications from each other. Neither do they provide the ability to limit the resource consumption of applications. Consequently, the performance of current systems degrades severely in the presence of malicious or buggy code that exhibits ill-behaved resource usage. We show that Java runtime systems can be extended to support *processes*, and that processes can provide robust and efficient support for untrusted applications.

We have designed and built KaffeOS, a Java runtime system that provides support for processes. KaffeOS isolates processes and manages the physical resources available to them: CPU and memory. Unlike existing Java virtual machines, KaffeOS can safely terminate processes without adversely affecting the integrity of the system, and it can fully reclaim a terminated process's resources. Finally, KaffeOS requires no changes to the Java language. The novel aspects of the KaffeOS architecture include the application of a user/kernel boundary as a structuring principle for runtime systems, the employment of garbage collection techniques for resource management and isolation, and a model for direct sharing of objects between untrusted applications. The difficulty in designing KaffeOS lay in balancing the goals of isolation and resource management against the goal of allowing direct sharing of objects.

For the SpecJVM benchmarks, the overhead that our KaffeOS prototype incurs ranges from 0% to 25%, when compared to the open-source JVM on which it is based. We consider this overhead acceptable for the safety that KaffeOS provides. In addition, our KaffeOS prototype can scale to run more applications than running multiple JVMs. Finally, in the presence of malicious or buggy

This research was supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, Rome Research Site, USAF, under agreements F30602-96-2-0269 and F30602-99-1-0503.

This article is a condensation of G. Back's dissertation for the University of Utah School of Computing.

Authors' addresses: G. Back, Department of Computer Science, Virginia Polytechnic Institute and State University, 2160A Torgersen Hall, Blacksburg, VA 24061; email: gback@cs.vt.edu; W. Hsieh, Google, Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043; email: wilson.hsieh@gmail.com.

code that engages in a denial-of-service attack, KaffeOS can contain the attack, remove resources from the attacked applications, and continue to provide robust service to other clients.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Runtime environments*

General Terms: Algorithms, Design, Reliability, Languages, Performance

Additional Key Words and Phrases: Robustness, resource management, isolation, termination, language runtimes, virtual machines, garbage collection

1. INTRODUCTION

The need to support the safe execution of untrusted programs in runtime systems for type-safe languages is clear. Language runtimes are being used to execute untrusted code that may violate a system's safety or security. Current runtime systems implement *memory protection* in software through the enforcement of type safety [Bershad et al. 1995a]. They do not, however, sufficiently isolate untrusted code and are unable to control the computational resources used by such code. We describe KaffeOS, a design for a Java virtual machine (JVM) that allows for the robust execution of untrusted code. Like a hardware-based operating system that supports and manages multiple processes running on one physical machine, KaffeOS provides resource control and isolation to its processes.

Consider the following application scenarios that employ type-safe languages:

—Java applets enjoy widespread use. There is no prior trust relationship between the originator of an applet—who could be a malicious attacker—and the client who executes the applet. A Java program can be verified to ensure that the code will not compromise security on the client's machine, but it includes no defenses against denial-of-service attacks directed against computing resources such as memory and CPU. Even though Java was first released to the public in 1995, industrial browsers still do not withstand even the simplest of attacks [McGraw and Felten 1997].

—Java has also become popular for many server-side applications, such as Java Server Pages [Bergsten 2000] or Oracle's JServer environment [Lizt 1999]. Even though such code is usually trusted, a buggy application could cause the server to spend all its time collecting garbage and deny service to other applications.

—Extensible operating systems allow applications to download code into the kernel. In the SPIN extensible OS [Bershad et al. 1995b], both the kernel and the extensions are written in the type-safe language Modula-3, which enables extensions' access to kernel interfaces to be controlled. However, it is impossible to control the resources used by a given extension—for instance, to guarantee that one extension obtains a certain share of CPU time.

These applications require a runtime system that supports the following features:

Protection. Protection includes confidentiality and integrity. Confidentiality requires that an application must not be able to read another application's data unless permitted. Integrity requires that an application must not be able to manipulate the data of another application or system-level data in an uncontrolled manner, or destroy the data of another application.

Isolation. Applications must be isolated from each other. One application's failure must not adversely affect unrelated applications or the system itself.

Resource management. First, resources allocated to an application must be separable from those allocated to other applications to ensure proper accounting. Second, resource management policies must guarantee that an unprivileged or untrusted application is not able to starve other applications by denying them resources.

Communication. Since the system may include multiple cooperating applications, applications should be able to communicate with each other. For Java, an efficient way of sharing data should be supported that does not compromise protection and isolation.

These features are provided by traditional operating systems through *processes*. They support the level of assurance that a system administrator would want to execute completely untrusted code. In certain situations, greater levels of trust may be permissible, in which case some of these features could be relaxed. However, in situations where untrusted code is to be executed, we believe that any relaxation would be undesirable.

Existing mechanisms such as type safety, language-based access control, and permission checks provide protection—our research shows how to support the remaining three features. Some existing systems [Czajkowski and von Eicken 1998; Hawblitzel et al. 1998; Gorrie 1998] provide limited support for these features. These systems superimpose an operating system model on Java, but do so without changing the underlying virtual machine. Figure 1(a) depicts the basic structure of these types of systems. Systems with this structure cannot account for resources that are spent by the JVM itself.

An alternative approach to separating different applications is to give each one its own virtual machine and run each virtual machine in a different process on an underlying OS [Jaeger et al. 1998; Malkhi et al. 1998], as shown in Figure 1(b). Depending on the operating system has multiple drawbacks: the per-JVM overhead is typically high, and the flexibility with which resources can be managed may be limited. For instance, a typical JVM's memory footprint is on the order of 1–2 MB, which can severely restrict scalability. A JVM's startup costs, which include the cost of loading and linking the Java bytecode, are typically high. When different instances of a JVM run on the same machine, they typically do not share any runtime data structures, even on systems that provide support for shared memory. Finally, the option of dedicating one JVM process to each application does not work on small devices that may not provide OS or hardware support for managing processes [Wind River Systems, Inc. 1995], or in software environments where the JVM is nested inside another application.

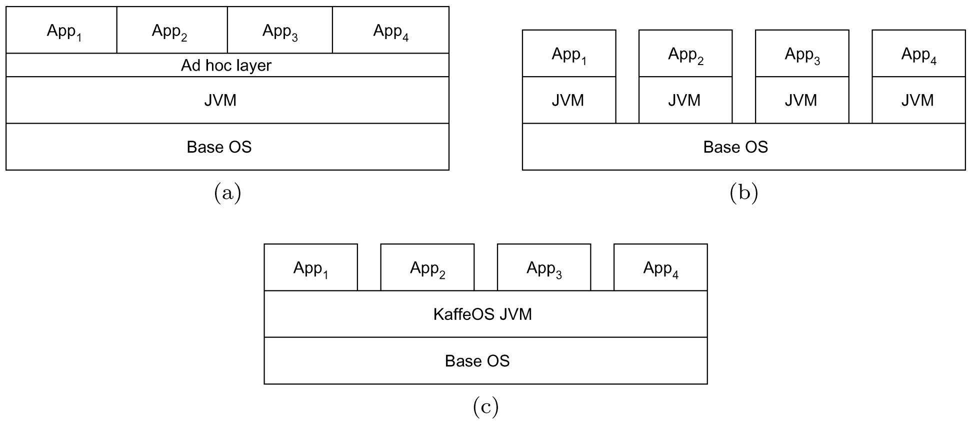


Fig. 1. Different JVM models. In the single JVM model, applications run on top of an *ad hoc* layer in one JVM. In the multiple JVM model, each application runs in its own JVM as a separate process on top of the underlying base operating system. By supporting multiple processes within the JVM itself, KaffeOS achieves both the efficiency and ease of sharing of the single JVM model and the isolation provided by the multiple JVM model.

Our approach to providing isolation and resource management is shown in Figure 1(c). It uses the same basic principles upon which operating systems are built. On a traditional operating system, untrusted code can be executed in its own process; CPU and memory limits can be placed on the process; and the process can be killed if it is uncooperative. We have designed and built a runtime system for Java that supports such a *process* abstraction. The mechanisms we describe are not specific to Java and are applicable to other type-safe languages.

Our design has three advantages over the other designs. First, it reduces per-application overhead. For example, applications can share runtime code and data structures in much the same way that an OS allows applications to share libraries. Second, communication between processes can be more efficient in one VM. Processes can share data directly through a direct memory reference to a shared object. Direct sharing is more efficient than exchanging data by copying it or sharing it indirectly through intermediate proxy objects. Third, a JVM that does not rely on underlying operating system support can be used in environments where such support is missing. For example, such a JVM could be used on a portable or embedded device that may only have a minimal operating system, especially one that is not powerful enough to fully isolate applications; or it could be used when embedding one application in another.

Our design makes KaffeOS's isolation and resource control mechanisms comprehensive. We focus on the management of CPU time and memory, although other resources such as network bandwidth or persistent storage could be added in the future. We paid particular attention to memory management and garbage collection, which proved to be the issues that posed the largest challenges in designing KaffeOS. We have devised a scheme in which the allocation and garbage collection activities of different processes are separated, so that the memory consumption of individual processes can be separately accounted for and so that the garbage collector does not become a source of priority inversion.

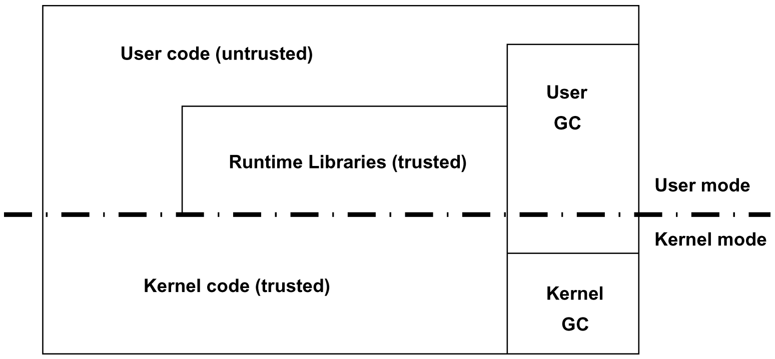
KaffeOS uses three different approaches for direct sharing of objects between processes. These approaches represent different points in the spectrum spanned by the conflicting goals of process isolation and resource management versus direct sharing. First, the sharing of kernel objects, which provide shared services, is possible without restrictions. Second, KaffeOS does not support the casual sharing of arbitrary objects between untrusted parties, because doing so would compromise isolation. Third, untrusted parties can share dedicated objects in specially created shared heaps. Shared heaps are subject to a restricted programming model so as not to compromise isolation. We demonstrate that our programming model for shared heaps is practical: despite its restrictions, applications can easily and directly share complex data structures in realistic applications.

To evaluate the feasibility of KaffeOS's design, we have built a prototype and analyzed its performance. In the case of trusted code, there is a small performance penalty for using KaffeOS. The run-time overhead ranges from 0% to 25%, relative to the JVM on which our prototype is based. We also demonstrate that our prototype can successfully thwart those denial-of-service attacks. KaffeOS can provide robust service to well-behaved applications, even in situations in which otherwise faster commercial JVMs provide practically no service at all. Finally, we compared KaffeOS to a configuration that is based on the multiple JVM model. We show that KaffeOS's performance scales better, and that it provides the same protection against misbehaved applications as an underlying operating system.

In Section 2, we discuss the principles underlying KaffeOS's design, and how they influenced the mechanisms we implemented. In particular, we discuss how KaffeOS reconciles the divergent needs of application isolation and sharing of resources and how it controls the resources used by its processes. We defer implementation-level details that are specific to our prototype to Section 3. This section describes the structure of the kernel, how the memory allocator and garbage collector work, and how the programming model for IPC works. Section 4 discusses the performance of our prototype, which includes both the overhead for trusted code and the performance improvement in the presence of untrusted code. Section 5 provides an in-depth discussion and comparison with related work. Section 6 suggests some directions for future work and summarizes our conclusions.

2. DESIGN

In this section, we describe how KaffeOS works: how we protect and isolate processes and how we guarantee safe termination; how garbage collection activities are separated; how interprocess communication (IPC) works; and how KaffeOS manages memory and CPU resources hierarchically. An important aspect of our design is that we wanted to avoid changing the programming model of Java whenever possible. As we will see, the only part of our design that impacts the programming model occurs with IPC.



	User Mode	Kernel Mode
Trusted	Runtime Code	Kernel Code
Untrusted	User Code	▨

Fig. 2. User/kernel boundary in KaffeOS. All system code, which includes the kernel, the runtime libraries and the garbage collector, can run either in kernel or user mode; such code enters kernel mode where necessary. By contrast, user code always runs in user mode. The bottom part of the figure shows that the trust dimension is orthogonal to the user/kernel mode dimension.

2.1 Protection and Isolation

KaffeOS depends on Java's type safety and language-based access control to provide some aspects of protection. First, Java's type safety guarantees that programs cannot forge pointers to obtain access to foreign objects. Second, KaffeOS uses Java's access modifiers, such as *private* and *protected*, to prevent processes from accessing fields in system objects to which they have access. Third, different processes use distinct classloaders, which provide each process with a namespace with a distinct set of types. Fourth, kernel code is not allowed to hand out references to a foreign process's objects or to internal system objects. For example, we must prevent a user process from acquiring a lock on an object that is used inside the kernel. We avoid leaking objects through interface design and careful coding; in the presence of bugs, language access modifiers and different namespaces would still restrict the possible use of such leaked objects.

In a hardware-based operating system, *user mode* and *kernel mode* indicate different CPU modes. User processes enter kernel mode through trap instructions when they execute system calls. A "red line" is said to separate user mode and kernel mode, which are different environments with respect to protection, termination, and resource control.

Figure 2 illustrates the high-level structure of KaffeOS. KaffeOS's modes represent a software analogue of the mode bit. The red line in KaffeOS, however, does not prevent user objects from directly accessing shared kernel objects. A

KaffeOS process can make system calls simply by invoking methods on kernel objects or classes. Threads executing these methods can in turn enter kernel mode by setting a software bit. User mode and kernel mode in KaffeOS are different environments with respect to termination and resource consumption:

—Resources consumed in user mode are always charged to a user process and not to the system as a whole. Only in kernel mode can a process consume resources that are charged to the kernel, although typically such use is charged to the appropriate user process.
—Processes running in user mode can be terminated at any time. Processes running in kernel mode cannot be terminated at an arbitrary time, because they must leave the kernel in a clean state. This design mirrors the way termination is handled in a hardware-based system such as Unix [Ritchie and Thompson 1978].

Application code executes in user mode, as do some of the trusted runtime libraries and some of the garbage collection code. The remaining parts of the system must run in kernel mode to ensure their integrity. These parts include the rest of the runtime libraries and certain parts of the virtual machine, such as the garbage collector or the just-in-time compiler. This structure echoes that of exokernel systems [Engler et al. 1995]. In KaffeOS the kernel can trust user-mode code to a great extent, because Java bytecode verification ensures that application code cannot violate the Java language semantics.

The KaffeOS kernel is structured so that it can cleanly handle both termination requests and resource exhaustion. Termination requests are deferred inside kernel code. Resource exhaustion is more difficult to handle; we adopt a kernel programming style that avoids the use of Java exceptions and uses explicit return code checking instead. This style is applicable to both code written in Java and in native code written in C. Avoiding Java exceptions also helps to remove the requirement that the exception handling facilities in the runtime system are not allowed to run out of resources.

Deferring termination to protect data structure integrity is superior to alternatives. In one alternative model, a termination request would be delivered as an asynchronous exception, and catch clauses would be added as cleanup handlers. Cleanup handlers in the form of *catch* clauses have a slightly lower cost than deferring termination in the common case, but programming them is tedious and error-prone. In addition, Marlow et al. [2001] has pointed out that the use of *catch* clauses for asynchronous exceptions can create race conditions. Another alternative solution, used in the Real-Time Java proposal [Bollella et al. 2000], defers asynchronous termination requests during synchronized sections. However, this alternative conflates termination and synchronization, which could lead to decreased parallelism, more deadlocks, or more confusing code.

In some situations, kernel code has to call out to user code. When making such upcalls, kernel code must be prepared for the user code to take an unbounded amount of time, or not to return at all, or to complete abruptly. The thread executing the upcall cannot hold on to system-level resources that may be needed by other processes prior to making the call. Other than through visual inspection, we do not currently have a means to ensure that. For some resources, static checking could be used to verify that they are released. For other resources, we could check this property at run time, but it would be hard to do in general unless we kept track of every lock a thread holds. Such bookkeeping would incur an overhead that would nullify the performance benefits of fast locking algorithms, such as thin locks [Bacon et al. 1998].

2.2 Memory Management

Each process is given its own heap on which to allocate its objects. In addition, there is a dedicated *kernel heap* on which only kernel code can allocate objects. Processes can allocate memory from their heaps without having an impact on other processes; in addition, they can garbage collect their heaps separately. Multiprocessor memory allocators such as Hoard [Berger et al. 2000] use per-CPU heaps to reduce lock contention and increase concurrency. In contrast, we use them for resource separation and accounting.

Our goal is to account precisely for the memory used by or on behalf of a process. Therefore, we account not only for objects at the Java level but for all allocations done in the VM on behalf of a given process. For instance, the VM allocates data structures during the loading and linking process for a class's symbols and range tables for exception handling. Accounting schemes based on bytecode rewriting [Czajkowski et al. 1998; Czajkowski and von Eicken 1998] can account only for object allocations, because they cannot modify the virtual machine. Accounting schemes based on garbage collection [Wick et al. 2002; Price et al. 2003] use a single garbage collector to estimate memory usage: they are imprecise[1] in the presence of sharing.

Kernel interfaces are coded so that memory that is used on behalf of a process is allocated on that process's heap. For example, a process object, which is several hundred bytes, is allocated on its associated heap (not the kernel heap). The handle that is returned to the creating process to control the new process is allocated on the creating process's heap. The kernel heap itself contains only a small entry in a process table.

We have designed KaffeOS to map logical resources to memory as much as possible. This design has the advantage that we need not create resource limits for every logical resource. For example, KaffeOS does not need specific limits for system objects such as open files, because it can allocate them completely on user heaps. Of course, some resources cannot be mapped to memory, and require explicit deallocation. For example, Unix network sockets must be closed using the `close(2)` system call. In KaffeOS, such resources are always associated with trusted system objects. Those objects must register reclamation handlers with the kernel, which are executed upon a process's exit.

2.2.1 *Reclamation.* We assume that all memory used by a process must be reclaimed when a process exits or is terminated. Complete reclamation prevents

[1]Wick et al. [2002] do describe a precise scheme, but the overhead of using their precise scheme is prohibitive.

the possibility of a *sharing attack*, where a process passes references to its own objects to another process in the hope that the second process can keep its own objects alive.

Reclamation must not sacrifice type safety, which implies that dangling pointers must not be created. Therefore, a process's objects must not be referenced from either another heap or a foreign thread's stack or registers. While a thread is in the kernel executing some system calls, it could have pointers to arbitrary user heaps. We designed the KaffeOS kernel so that system calls cannot block indefinitely if a termination request is pending; as a result, reclamation cannot be delayed indefinitely. In addition, we assume that a thread in one process that is not executing a system call cannot have pointers to another process's heap. That is, we designed the kernel so that it does not expose a process's heap to another process through system calls: such a leak would be a bug. Automatic program checking or verification techniques could be applied to enforce these rules statically [Engler et al. 2000; Chen and Wagner 2002; Henzinger et al. 2002].

Our kernel design restricts cross-process stack and register pointers. Cross-references between user heaps must be prevented, because such references would preclude reclamation of a heap. If we trusted that the kernel was bug-free, we could assume that cross-process heap pointers could not be created. Because maintaining isolation is a primary design goal, we do not make that assumption: KaffeOS monitors writes to the heap through the use of *write barriers* [Wilson 1992]. A write barrier is a check that happens on every write of an object reference into the heap. Unlike software-fault isolation schemes [Wahbe et al. 1993], we do not instrument every "store" machine instruction: write barriers need to be inserted only for instructions that store references to memory. In Java, these instructions are `PUTFIELD`, `PUTSTATIC`, `AASTORE`, and any assignment within the VM or in native libraries that creates a connection between two garbage-collected objects.

KaffeOS's write barriers prevent illegal cross-references by interrupting those writes that would create them and raising an exception instead. We call such exceptions "segmentation violations." Although it may seem surprising that a type-safe language runtime could throw such a fault, it actually follows the analogy to traditional operating systems closely.

2.2.2 *Garbage Collection.*

In KaffeOS, each process's garbage collection activity is independent. This independence enables KaffeOS to accurately charge processes for their CPU usage during garbage collection. The GC-based accounting schemes [Wick et al. 2002; Price et al. 2003] that we mentioned earlier do not have this feature, because they use one pass to garbage collect all objects. As a result, such schemes account for garbage collection time with the same precision with which they measure the memory consumption of shared objects.

KaffeOS makes use of techniques from distributed garbage collection [Plainfossé and Shapiro 1995] to track cross-references between heaps. Write barriers detect the creation of legal cross-references. When a cross-reference is created, we create an *entry item* in the heap to which it points. *Exit items* in the original heap keep references to corresponding entry items. Exit items

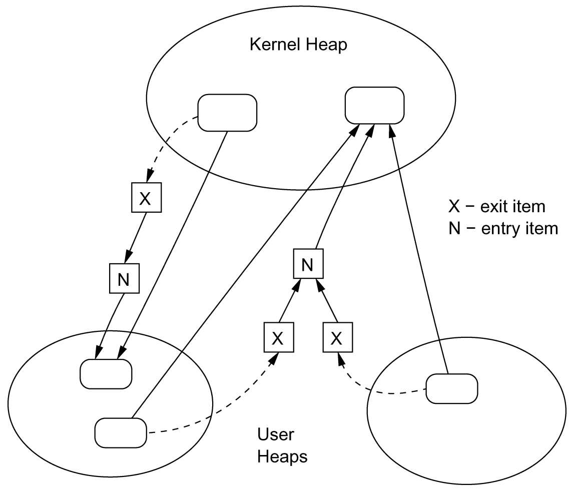


Fig. 3. Use of entry and exit items. Legal references between user heaps and the kernel heap are monitored through exit and entry items. Multiple exit items can refer to the same entry item if an object is shared by more than one heap. Dashed lines mean that objects do not need to store a direct pointer to the exit items; instead, they can be found through a hashtable indexed by the remote address.

can share entry items, as shown in Figure 3. Unlike distributed object systems such as Emerald [Jul et al. 1988], entry and exit items are not used for naming nonlocal objects. Instead, they are used to decouple the garbage collection of different heaps.

During a garbage collection cycle, entry items are treated as garbage collection roots. Entry items are reference counted: they keep track of the number of exit items that point to them. When an entry item's reference count reaches zero, the entry item is removed, and the referenced object can be garbage collected if it is not reachable through some other path. Exit items are subject to garbage collection: if the garbage collector encounters a reference to an object outside the current heap, it will not traverse that object, but the corresponding exit item instead. Exit items that are not marked at the end of a GC cycle are garbage collected, and the reference count of the entry item to which it points is decremented.

Because cross-references between heaps are reference counted, we must pay particular attention to cycles. Cycles cannot occur between user heaps; they can only occur between the kernel heap and user heaps. Writes to kernel objects can only be done by kernel code; therefore, we can control what cycles are created, modulo any bugs. Our kernel code tries to avoid cycles where possible. However, some cycles are unavoidable: for instance, a process object is allocated on the user heap, while its process table entry is allocated on the kernel heap. These two objects must reference each other. Our kernel allows such cycles only if they include system objects whose expected lifetime is equal to the lifetime of their owning processes.

User-kernel heap cycles are collected when a process terminates or is killed. After we destroy all its threads, we garbage collect the process's heap and merge the remaining objects into the kernel heap. Garbage collection of the kernel heap can then collect any unreachable cycles. Because we allow user-kernel cross-references, it is theoretically possible that a cycle that spans multiple heaps could be created. Such a cycle would require intermediate objects on the kernel heap that connected different user heaps, which would be considered a kernel bug.

2.3 Interprocess Communication

In standard Java sharing is *direct*: objects contain pointers to one another, and a thread accesses an object's fields via offsets from the object pointer. Since Java is designed to support direct sharing of objects, we choose to allow direct sharing *between* processes. Direct sharing in single-address-space systems is somewhat analogous to shared memory (or shared libraries) in separate-address-space systems, but the unit of sharing is at a finer granularity.

Direct sharing between processes complicates process termination and resource reclamation. If a process exports a directly shared object arbitrarily, it is possible that object cannot be reclaimed when the exporting process is terminated. All exported references to a shared object must remain valid, so as not to violate the type-safety guarantees made by the Java virtual machine. As in the case of user/kernel sharing, we suitably restrict references while still maintaining most of the benefits of direct sharing. Unlike user/kernel sharing, however, our direct sharing model does not rely on trust that the sharing parties avoid exporting references that would prevent reclamation. In addition, the model guarantees that the amount of shared memory used by a given process is known at all times.

In KaffeOS, a process can dynamically create a shared heap to directly share objects with other processes. Our design for shared heaps guarantees three properties. First, one process cannot use a shared heap to keep objects in another process alive. Second, all sharers are charged in full for a shared heap while they are holding onto the shared heap, whose size is fixed. Third, sharers are charged accurately for all metadata, such as class data structures.

Shared objects are not allowed to have pointers to objects on user heaps, because those pointers would prevent the user heap's full reclamation. This restriction is enforced by our write barriers; attempts to create such pointers result in an exception. Figure 4 shows which references are legal between user heaps, shared heaps, and the kernel heap. References between shared heaps and the kernel heap are legal, similar to references between user heaps and the kernel heap.

Shared resources pose an accounting problem: if a resource is shared between n sharers, should all n sharers be charged $1/n$ of the cost? In such a model, the required contribution of each sharer would grow from $1/n$ to $1/(n-1)$ when a sharer exits. Sharers would then have to be charged an additional $1/(n-1) - 1/n = 1/n(n-1)$ of the total cost. As a result, a process could run out of memory asynchronously through no action of its own. Such behavior would violate isolation between processes. Therefore, we charge all sharers in full for shared data when they obtain references to it and reimburse sharers in full when they are done using the shared data.

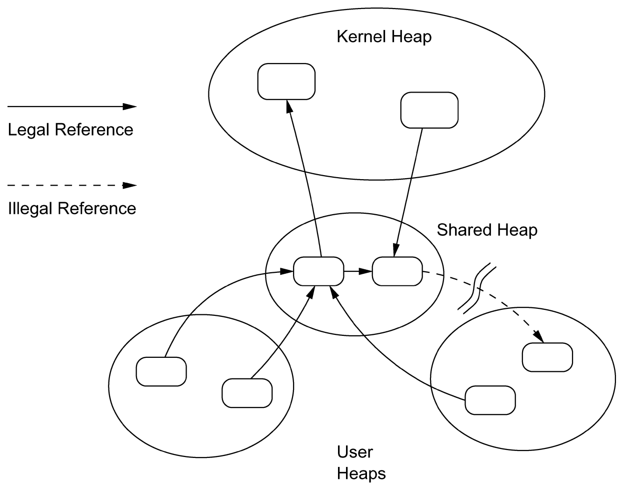


Fig. 4. Valid cross-references for shared heaps. References from a user heap to a shared heap are legal and necessary to share objects. References from a shared heap to any user heap would prevent full reclamation and are therefore illegal. Shared objects may refer to kernel objects and vice versa.

A similar problem would occur if one party could expand the size of the data that is shared. If the size of the shared data increased over time, a process could asynchronously run out of memory. To avoid this problem, we allow processes to share only data of fixed size. This decision may lead to some waste of memory, because applications that do not know how much data they will share may have to overprovision the shared heaps they use.

Processes can exchange data by accessing shared objects, which reside on shared heaps. Acquiring references to shared objects requires the invocation of a special API, but making use of the objects does not. Instead of introducing a special API, another possibility would be to change Java itself. For example, one could provide a modified form of the `new` operator that would take the heap on which to perform the allocation as a parameter.

A shared heap has the following lifecycle:

(1) One process picks one shared class out of a central shared namespace, creates the heap, loads the shared class into it, and instantiates a specified number of objects of that shared class. During this instantiation, each object's constructor is invoked using the Java reflection API. All memory allocated during this process is charged to the newly created shared heap. A shared class can execute arbitrary code in its constructor, which includes instantiating objects of additional classes. While the heap is being created, its memory is charged to its creator. The creating process does not need to know the actual size of the shared objects in advance.

(2) After the heap is populated with classes and objects, its size is frozen. Kernel code keeps track of what shared heaps are currently in use and provides a lookup service to user processes. Other processes can use this service to look up a shared heap; this operation returns a direct reference to the objects in it. When a process looks up a shared heap, it is charged the amount

established when that heap was frozen. The lookup fails if the amount exceeds the potential sharer's memory budget. Our prototype does not provide access controls on shared heaps, although they could be added easily.

(3) When a process drops all references to a shared heap, all of its exit items to that shared heap are unreachable. After the process garbage collects the last exit item to a shared heap, that shared heap's memory is credited to the sharer's budget. When the last sharer drops all references to a shared heap, the shared heap can be garbage collected. The number of shared heaps a process can create during a given time period is limited; otherwise, a process could repeatedly create and abandon shared heaps and deny service to other processes.

Metadata related to shared classes and objects are allocated on the shared heap. Because the shared heap is frozen after it is populated with objects, we need to ensure that no further allocations of metadata are necessary afterwards. For this reason, we eagerly perform allocations that would otherwise be performed lazily in Java, such as the compilation of bytecode to native code and the resolution of link-time references.

Allocations on a frozen shared heap and writing references to user-heap objects into shared objects triggers segmentation violation exceptions. Therefore, certain dynamic data structures, such as variable-length queues or trees with a dynamically changing number of nodes, cannot be efficiently shared. Their fixed-sized counterparts can be shared, however. For instance, it is easily possible to share such data structures as read-only tries that are used for dictionary lookup.

Our direct sharing mechanism does not require the kernel to trust the code associated with any shared classes. Untrusted parties, without additional privileges, can share data. However, the communicating parties must trust each other: our sharing mechanisms do not defend a server against an untrusted client. If an untrusted client is terminated while operating on a shared object, the shared object might be left in an inconsistent state.

We could easily extend KaffeOS to permit trusted servers to enter kernel mode to protect shared data structures. In this manner, we could implement trusted servers that communicate with untrusted clients. Such an extension would be analogous to supporting in-kernel servers in microkernel-based systems [Lepreau et al. 1993]. If a server could not be trusted in this way, clients would have to use one of the traditional, copy-based communication mechanisms Java supports, such as sockets or RMI. Alternatively, kill-safe abstractions could be used [Flatt and Findler 2004].

Our IPC mechanism does not satisfy all of Shapiro's requirements for assured IPC [Shapiro 2003]: asymmetric trust, reproducibility, and dynamic payload. KaffeOS IPC is based on object invocation, which cannot satisfy the requirement of asymmetric trust: the ability to pass closures between clients and servers requires mutual trust. KaffeOS IPC is reproducible, because its semantics does not depend on the system's status. Finally, KaffeOS IPC supports dynamic payload, because a server object on a shared heap can allocate memory in the client's heap. Note that IPC across the user-kernel boundary does satisfy all three requirements: the kernel is designed to distrust any closures provided from user space.

If we modified our assumptions to relax our requirements, we could adjust some aspects of our model in special cases. For instance, if we can rule out sharing attacks because all sharers are known to terminate at about the same time, we could allow some cross-references from shared heaps to user heaps. Second, if the communicating processes trusted each other and shared a common soft ancestor memlimit (described in the next section), we might not have to freeze the shared heap and charge all sharers. Instead, we could associate the shared heap with a sibling memlimit and leave the responsibility to avoid out-of-memory situations to the sharing processes.

GC-based accounting schemes take approaches that cannot ensure the same precision in accounting for GC time. Wick et al. [2002] proposes a scheme in which child processes are scanned before parent processes, which ensures precise accounting only under certain kinds of sharing patterns. (Their precise scheme incurs unacceptable performance penalties.) Price et al. [2003] proposes a scheme in which roots are scanned in different orders, which is accurate only if sharing patterns are stable over time.

2.4 Hierarchical Resource Management

In KaffeOS, the kernel manages primary resources. Primary resources are those that must be shared by all processes, such as CPU time, memory, or kernel data structures. Kernel data structures can be managed simply by applying conventional per-process limits. In this section, we focus on the mechanisms for setting resource management policies for memory and CPU time.

To manage memory, we associate each heap with a *memlimit*, which consists of an upper limit and a current use. The upper limit is set when the memlimit is created and remains fixed for its lifetime; only the current use portion is updated. Memlimits form a hierarchy: each one has a parent, except for a root memlimit. All memory allocated in a heap is debited from its memlimit, and memory collected from a heap is credited to its memlimit. This process of crediting/debiting is applied recursively to the node's parents, according to the following rules.

A memlimit can be *hard* or *soft* (note that a soft memlimit is not one that can be exceeded temporarily, as in filesystem quotas). This attribute influences how credits and debits percolate up the hierarchy of memlimits. A hard memlimit's maximum limit is immediately debited from its parent, which amounts to setting memory aside for a heap. Credits and debits are therefore not propagated past a hard limit after it has been established. For a soft memlimit's maximum limit, on the other hand, no memory is set aside, so a process is not guaranteed to be able to allocate its full limit. It cannot, however, individually allocate more than this limit. All credits and debits of a soft memlimit's current usage are immediately reflected in the parent. If the parent is a soft limit itself, the process is applied recursively such that an allocation fails only if it would violate the maximum limit of any ancestor node.

Hard and soft limits allow different memory management strategies. Hard limits allow for memory reservations but incur inefficient memory use if the

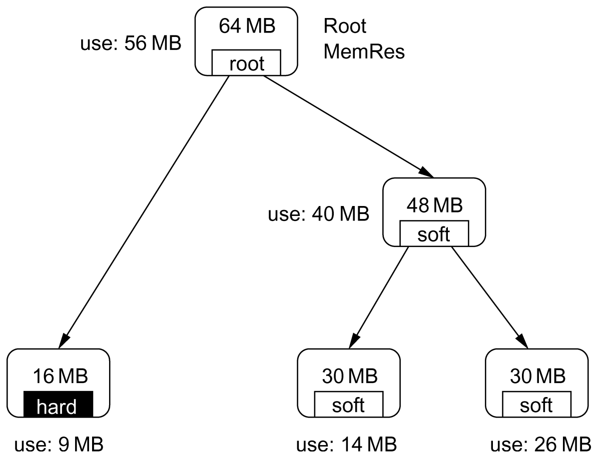


Fig. 5. Hierarchical memory management in KaffeOS. The numbers inside the boxes represent limits; the numbers outside represent current use. A hard limit is immediately debited from its parent; for soft limits, only the actual use is debited.

limits are too high. Soft limits allow the setting of a *summary* limit for multiple activities without incurring the inefficiencies of hard limits, but they do not reserve memory for any individual activity. They can be used to guard malicious or buggy applications where high memory usage can be tolerated temporarily.

Consider the example shown in Figure 5: the 64-MB root memlimit is split into a hard memlimit with a total of 16 MB on the left, of which 9 MB are used, and a soft memlimit with a total of 48 MB on the right. The soft memlimit is further subdivided into two soft memlimits with a total of 30 MB each. Because these memlimits are soft, the sum of the maximum limits of sibling nodes can nominally exceed the parent's maximum limit. Doing so allows a child to use more than it would have been able to use had the memory been strictly partitioned.

We use a stride scheduler [Waldspurger 1995] for CPU scheduling of processes. We do not specify how threads within a process should be scheduled: the Java specification demands a priority-based scheduler for the threads within one application [Joy et al. 2000, Sect. 17.12]. Our CPU resource objects represent a share of CPU. Stride scheduling is work-conserving: CPU time that a process does not use can be used by other processes. Therefore, it does not limit how much CPU time a given process uses under light load. Under full load, if all processes are runnable, a process uses only its assigned share. The kernel heap's collector and finalizer threads are the only threads not scheduled by the stride scheduler: they are always scheduled whenever they are runnable. As already mentioned, denial-of-service attacks against the kernel heap can be prevented by limiting the number of kernel operations a process can perform.

3. IMPLEMENTATION

Our KaffeOS prototype is an extension of Kaffe [Wilkinson 1996], an open-source Java VM. It is not a very fast JVM because its just-in-time compiler performs little to no optimizations. However, Kaffe is mature and complete enough to run real-world applications. While our design does not rely on any specific properties of Kaffe, our prototype depends heavily on the details of Kaffe's implementation.

3.1 Kernel

We implemented the KaffeOS kernel as a set of Java classes with corresponding native methods. Implementing the kernel involved implementing classes for process and heap management, and introducing the red line to ensure safe termination. We introduced the red line by redesigning some sections of code and by applying transformations to others so that critical data structures are manipulated in kernel mode and so that kernel code handles all exceptional situations carefully. This experience shows that the user/kernel boundary can be introduced into an existing code base; nevertheless, it should be used as a structuring principle from the beginning when designing JVMs or other runtime systems.

We implemented basic kernel services as classes in a package `kaffeos.sys.*`. We implemented as many methods as possible in Java and resorted to native methods only when necessary, mainly for glue code to the portions of the virtual machine written in C. Most of the kernel-related services are implemented in the classes `Kernel`, `Heap`, and `Process`. Other kernel services, such as opening a file or network socket, are implemented by adapting the existing classes in the Java runtime libraries, such as `java.net.Socket`. Finally, we changed all kernel code so that it does not synchronize on objects that are exposed to user code.

`Kernel` implements basic functionality to bootstrap and shutdown the system or to enter and leave kernel mode; it also provides methods to control various internal kernel properties. `Heap` implements access to the (native) implementation of multiple heaps; it allows for the creation of new heaps and provides methods to manipulate existing heaps. It also provides functionality for deep copying of objects between heaps. Deep copies are necessary when arguments to a system call must be passed from one process to another. For instance, a process's command line arguments must be deep copied, because a child process is not allowed to refer directly to the arguments stored in its parent process. Only objects of known classes, such as file descriptors or immutable objects such as strings can be deep copied; we do not support deep-copying of arbitrary user-defined objects.

The `Process` class provides the following functionality:

—bootstrapping of new processes,

—code to safely kill a process and invoke any necessary cleanup handlers to free its logical resources,

—handles that can be used to control running processes,

—process-local properties, such a timezone or language-specific output settings.

3.2 Namespace and Class Management

We use Java class loaders to provide KaffeOS processes with different namespaces. Other language-based systems provide different namespaces in other ways: for example, the SPIN project used a mechanism called domains, implemented by a dynamic linker [Sirer et al. 1996]. Java represents its runtime types as class objects, and each class object is distinguished by its defining class loader. We use class loaders in two ways. First, we use multiple loaders if we cannot allow or do not want processes to share classes. Second, we delegate to a common loader for processes that share classes.

Reloading Classes. *Reloading* is the multiple instantiation of a class from identical class files by different loaders. Although these instantiations are of different types, they use the same code, and thus their visible behavior is identical. Reloaded classes are analogous to traditional shared libraries. Reloading a class gives each instance its own copies of all static fields, just as a shared library uses a separate data segment in each process in which it is mapped. Reloaded classes could share text, although our current implementation does not support such sharing. Sharing text could be accomplished by changing the just-in-time compiler to emit code that follows shared-library calling conventions. Such code would use indirect calls through a jump table for calls that lead outside the library's code segment.

Sharing Classes. We can share classes between different processes' namespaces by delegating requests for them to a common loader. We share classes in two cases: either because the class in question is part of the runtime library or because it is the type of a shared object located on a shared heap. We refer to the former as system-shared, and the latter as user-shared. System sharing of classes makes more efficient overall use of memory than reloading them, because only a single class object is created. Even if our implementation shared the text and constants of reloaded classes, the advantage of having system-shared classes would likely still be significant, because a system-shared class needs to maintain only a single copy of its linking state. However, system-shared classes are not subject to per-process accounting, because we assume that such sharing benefits all processes. Their number and size are bounded, because applications cannot add system-shared classes. Our goals in developing our class loading policy were to gain efficiency by maximizing system sharing and to maintain correctness and ease of use for user-shared classes. At the same time, we had to take into account the sometimes conflicting constraints imposed by the Java API and Kaffe's linker implementation, which we discuss below.

Process Loaders. Each KaffeOS process has its own class loader, which manages that process's namespace. The process loader is a trusted system object that implements the loading policy. When asked for a class, it decides whether the class is shared or not. If so, the request is delegated to either the system loader (for system-shared classes), or a shared loader (for user-shared classes). If the class is not shared, it is either a system class that must be reloaded, or a regular user class. For user classes, our default implementation uses the standard Java convention of loading classes from a set of directories and compressed archives that are listed in a `CLASSPATH` variable.

To ensure full namespace control, the process loader must see all requests for classes, including those from user-created class loaders in that process. By default, user-created class loaders in standard Java first attempt to delegate to the system loader to ensure that system classes are not replaced by user classes. We changed the class loader implementation to delegate to the process loader whenever a user-created loader would delegate to the system loader in a conventional JVM.

The class loading mechanism must ensure that dynamic loading of classes does not violate type safety, even if user-defined class loaders behave maliciously or contain bugs. The class loader algorithm and implementation in early versions of Java was found to be faulty [Saraswat 1997; Dean 1997]. In response, a set of rules that govern the behavior of user-provided class loaders was developed [Dean 1997; Liang and Bracha 1998]. These constraints are orthogonal to the restrictions related to KaffeOS's process model. While our prototype does not implement these rules, an implementation would not interfere with providing per-process namespaces.

System-Shared Classes. To determine which classes can be system-shared, we examined each class in the Java standard libraries [Chan et al. 1998] to see how it acts under the semantics of class loading. In particular, we examined how classes make use of static fields. Because of some unfortunate decisions in the Java API design, some classes export static fields as part of their public interfaces. For example, `java.io.FileDescriptor` exports the public static variables `in`, `out`, and `err` (which correspond to stdin, stdout, and stderr.) Other classes use static fields internally. To system-share classes with static fields, we must either conclude that it is safe to leave the fields unchanged; we must eliminate the fields; or we must initialize the fields with objects whose implementation is aware of KaffeOS's process model. Final static fields with a primitive type such as `int` are constants in Java and can be left untouched. Elimination of static fields is sometimes possible for singleton classes, which can be made to use object fields instead of static fields. If static fields cannot be eliminated, we examine whether they can be initialized with objects whose implementation is process-aware. If an object stored in a static field is used only by invoking its methods, we can provide a substitute implementation that provides the same methods, but modifies their behavior to take the currently executing process into account. For instance, a process-aware version of the `System.out` stream maintains a stdout stream for each process and dispatches output to the appropriate stream.

If the object's fields are directly accessed by other classes, then creating a substitute implementation becomes more complicated, because it requires changes to the code that uses that object. As a practical matter, we tried to make as few code changes as possible, to allow for easy inclusion of upgrades and bug fixes developed for the code base upon which KaffeOS is built. Reloading classes allows us to use a class with static fields practically unchanged in a multiprocess environment. Consequently, we reload some classes that could have been rewritten to be shared.

Linker Constraints. The decision of whether to share or reload a class is subject to linker-specific implementation constraints as well. First, shared classes cannot directly refer to reloaded classes, because such references are represented using direct pointers by the run-time linker, and not through indirect pointers as in shared libraries. Second, certain classes must be shared between

```
public final class System {
    final public static PrintStream out;

    static {
        // direct reference to reloaded FileDescriptor class
        FileOutputStream f = new FileOutputStream(FileDescriptor.out);
        BufferedOutputStream b = BufferedOutputStream(f, 128);
        out = new PrintStream(b, true);
    }
}
```

```
/* KaffeOS version */
public class ProcessStdoutPrintStream extends PrintStream {
    /* process-aware version */
    void println(String s) throws IOException {
        Process.getCurrentProcess.getStdout().println(s);
    }
}

public final class System {
    final public static PrintStream out = new ProcessStdoutPrintStream();

    private static void kaffeos_init() {
        Class fdclass;
        fdclass = Class.forName("java.io.FileDescriptor", true,
                                Process.getRootLoader());
        Field f = fdclass.getField("out");
        ((ProcessStdoutPrintStream)out).createStdoutStream(f.get(null));
    }
}
```

Fig. 6. Transforming library code. The original version of `System.out` was implemented as a simple `PrintStream` layered on top of a buffered file stream, which in turn relied on a `FileOutputStream` constructed using the stdout file descriptor `FileDescriptor.out`. Because `System` must be shared, `System.out` is replaced with a process-aware version that dispatches output to the current process's stdout stream. As in the original version, this stream is initialized with the current process's `FileDescriptor.out`. Because `FileDescriptor` is reloaded, reflection is used to avoid a direct reference from `System` to `FileDescriptor`.

processes. For example, the `java.lang.Object` class, which is the superclass of all object types, must be shared. If this type were not shared, it would not be possible for different processes to share generic objects! Consequently, the transitive closure of classes pointed to by `Object` has to be shared.

We built a small tool that determines the transitive closure of classes referenced by a class, and used the tool on the classes we needed to share. For `java.lang.Object`, we found that its closure turns out to be quite large: `Object` refers to `java.lang.Throwable`, which is the base class of all exceptions, which in turn refers to `java.lang.System` in its `printStackTrace` method. `System` is a complex class with a multitude of references to other classes. We then rewrote classes to remove direct references. This transformation requires the use of reflection to access fields or methods in a class. An example of such a transformation is given in Figure 6, which depicts the initialization code sequence for the `System` class. In the original version, the `out` variable was implemented as a simple `PrintStream` layered on top of a buffered file stream, which in turn relied on a `FileOutputStream` constructed using the stdout file descriptor `FileDescriptor.out`. `System.out` is replaced with a process-aware object that dispatches the output to the current process's stdout stream; `println()` is shown as an example. The current process's output stream is initialized with the current process's `FileDescriptor.out` in a method called `kaffeos_init()`. Because `FileDescriptor` is reloaded, reflection is used to resolve the name `java.io.FileDescriptor` in the context of the current process's loader, which avoids a direct reference from `System` to `FileDescriptor`. By convention, the static `kaffeos_init()` method, if it exists, is invoked for every shared class at process initialization time. It serves as an analogue to static initializers in reloaded classes. Code that would otherwise be in a static initializer is moved to `kaffeos_init()` if process-local data must be initialized.

Our use of the term *reloading* differs from the use in Liang and Bracha [1998], where it describes a technique to dynamically update systems by loading multiple versions of the same class over time. Instances of classes reloaded in this way can be accessed directly through a known shared supertype. In this case, the linker does not have to create a direct connection between the class accessing such instances and the reloaded classes themselves, because the accessing class only refers to the shared supertype. KaffeOS supports this dynamic update technique for its applications as well. The need for the indirection discussed in this section arises only if the Java API dictates a direct relationship between classes that must be shared and classes that cannot be shared, as is the case in the example discussed above.

When writing native methods for reloaded classes, we must ensure that references to class objects and static fields are properly handled. In standard Java, a native library can be loaded only by a single class loader [Liang 1999, Sect. 11.2.4]. This restriction was introduced to prevent native code from mistakenly intermixing classes and interfaces from different class loaders. Such intermixing could occur if native libraries cached pointers to class objects and static fields between calls. Such caching is safe if classes that contain native methods are not reloaded. KaffeOS does not impose this restriction and reloads classes with native methods. Therefore, we must ensure that its native libraries do not perform such caching.

Adapting the runtime libraries to KaffeOS was a compromise between our desire to share as many classes as possible to increase efficiency, yet at the same time exploit reloading to avoid code changes. Out of roughly 600 classes in the core Java libraries, we are able to safely system-share 430. This high proportion (72%) was achieved with only moderate changes to a few dozen classes and more extensive changes to a few classes, such as `java.lang.System`. The rest of the classes are reloaded, which requires no changes at all. The set of runtime classes that an application uses is, unsurprisingly, application-dependent. Table I shows the distribution for the runtime classes used by the SPEC JVM98 benchmarks. The proportion of shared classes lies above 72%, although these benchmarks do not exercise many runtime classes, and the results may therefore not be representative.

Table I. Number of Shared Runtime Classes for SPEC JVM98 Benchmarks

Benchmark	check	compress	jess	db	javac	mpegaudio	mtrt	jack
reloaded	8	8	8	8	8	8	8	8
shared	55	46	54	49	60	49	51	47

This table shows that of the runtime classes used by the SPEC JVM98 benchmarks, only a small number cannot be shared and must be reloaded. For all benchmarks, `java.io.FileDescriptor`, `java.io.FileInputStream`, and `java.io.FileOutputStream` are among the eight reloaded classes.

User-Shared Classes. To ensure that every process that has access to a shared heap sees the same types, process loaders delegate the loading of all user-shared types to shared loaders. Each shared heap has its own shared loader, which is created at the same time as the heap. Process loaders use a shared class's name to determine the shared loader to which the initial request for a shared class should be delegated. Java's class loading mechanism ensures that subsequent requests are delivered to the shared loader that is the defining loader of the initial class. If we did not delegate to a single loader, KaffeOS would need to support a much more complicated type system for its user-shared objects.

For simplicity, all user-shared classes share a single global hierarchical namespace in `shared.*`, similar to a filesystem. Communicating partners can look up shared classes by name. A table maps class names to loaders. This table is a global resource, and limits on the number of entries a process can create could be applied to prevent that table from growing indefinitely.

Like system-shared classes, user-shared classes cannot directly refer to reloaded classes. Because most classes that are part of the runtime library are shared, this limitation is not severe. As with system-shared classes, a developer of user-shared classes can use reflection to access reloaded classes. Failure to use reflection results in write barrier violations during linking, because the linker would attempt to create a cross-reference from a class on a shared heap to a class on a user heap.

3.3 Memory Management

We focus on the *process-specific* implementation aspects of the memory management subsystem of our KaffeOS prototype. Most of the other implementation details were relatively straightforward. The implementation consists of two parts: the memory allocator and the garbage collector.

3.3.1 *Memory Allocator.* The memory allocator is logically divided into two components: the small object allocator and the primitive block allocator. The small object allocator maintains a pool of pages from which small objects are allocated. Each heap has its own pool of small object pages. The primitive block allocator maintains a single global pool of continuous memory regions that are comprised of one or more contiguous pages. Objects that are larger than one page and small object pages are allocated from the pool of primitive blocks.

Keeping the free lists for small objects on a per-heap basis has the advantage that all internal fragmentation is fully accounted for. Once a page is taken from the primitive block list and prepared for use in a given heap, it is charged to that heap. As a result, it is impossible for a process to launch a "fragmentation attack" against KaffeOS by holding on to a few objects that are spread over multiple pages. A second advantage is that no cross-process locking is necessary during allocation. Finally, because all objects within one page belong to the same heap, we need to store the heap identifier only once for all objects on a page.

Conversely, because a page will be charged to a heap even if only a single object is used, processes may be overcharged for memory. If we consider only live objects, we can expect an average overcharge of $n * pagesize/2$, where n is the number of freelists. For $n = 19$ small object sizes, the overhead amounts to 38 KB on a machine with 4 KB pages. Picking a smaller n would reduce this overhead, but would increase the slack that is wasted per object. Internal fragmentation can also lead to an overcharge when a heap is not garbage collected frequently enough, so that it allocates long-lived objects in new blocks instead of garbage collecting and reusing older blocks. The possible impact of this overcharge is difficult to predict, because it depends on an application's allocation and garbage collection patterns. Note, however, that overcharging because of internal fragmentation does not require an application to have a memlimit with a maximum limit greater than the maximum total size of its live objects, because a garbage collection is triggered when a heap reaches its maximum limit. This fragmentation issue occurs only because our collector does not move objects, and it is not inherent in our design.

Although internal fragmentation can be accounted for, our implementation is subject to external fragmentation of global memory. In practice, this fragmentation caused problems with lazily allocated kernel data structures, because scattered small object pages with long-lived kernel data fragmented the space of primitive block, which reduced the maximum size of primitive blocks available. To prevent the kernel from causing external fragmentation in this way, we adopted a work-around: when possible, we allocate kernel pages from a large continuous block at the bottom of the global memory pool. The primitive block allocator allocates blocks of memory whose sizes are multiples of a pagesize, and external fragmentation can occur. Like the internal fragmentation discussed earlier, this external fragmentation could be avoided if a moving collector were used.

3.3.2 *Garbage Collector.* KaffeOS uses the garbage collector provided by Kaffe, which is a nongenerational, nonincremental collector that walks stacks conservatively. For correctness, we must ensure that the set of objects that we identify as reachable does not change during the mark phase, even if the reachability graph should change. The simplest solution would be to prevent all threads from running during the full GC cycle. This solution would violate the separation of different processes if we prevented all remote threads from running for the full duration of the collection.

Given the absence of fully incremental garbage collection in Kaffe, we chose to implement a simpler solution. In our solution, we block those foreign threads that attempt writes into a heap when a garbage collection for that heap has been started. Usually, this blocking affects few, if any, threads. Only foreign threads in kernel mode have a legitimate reason to write into a process's heap, and only if they perform kernel operations directly related to that process. Examples of such operations include a process starting or stopping another process, or retrieving a process's properties.

Our thread stack algorithm needs to perform more work than in the single process case. In particular, it needs to walk each remote entry twice, as well as all objects reachable from them. Because entry items to user heaps can be created only by kernel code, there are generally few of them. For instance, a simple "Hello, World" program creates only two entry items. The collection of the kernel heap itself is a special case, where we stop all threads for the entire duration of GC.

3.3.3 *Implementing KaffeOS on Modern VMs.* The implementation of KaffeOS design would likely be significantly different on a modern VM. We can envision several ways in which the implementation could benefit:

Fast Write Barriers. We would combine KaffeOS write barriers with the write barriers used by the VM's collector. For instance, some allocators divide the heap into heap regions that are aligned at major virtual memory alignment boundaries [Stefanović et al. 1999]. Checking whether the source and target of an assignment are within the same heap region—and therefore, in KaffeOS, belong to the same KaffeOS heap—can be done without loading explicit bounds. This common case takes only three or four instructions on most architectures.

Fully Precise Collection. Using a fully precise collector with garbage collection maps would help us in at least two ways. First, we would be immune against certain kinds of denial-of-service attacks against the garbage collector that are based on spoofing pointers on the stack to which a conservative collector is vulnerable. Second, a precise collector would allow us to identify which remote threads contain kernel frames on their stack, aiding the process of scanning foreign threads.

Pretenuring. A generational collector could employ application-specific knowledge for pretenuring [Blackburn et al. 2001]. In particular, shared kernel objects that are known to be long-lived can be directly allocated in regions that are least frequently collected. This would reduce the overhead associated with collecting the kernel heap.

Thread-Local Heaps. Thread-local heaps [Steensgaard 2000; Domani et al. 2002] are used to reduce the synchronization overhead in a multi-threaded environment. Each thread is given a heap in which to allocate objects that are not shared with other threads, determined by performing a escape analysis. KaffeOS could benefit because assignments to objects known to not escape the current thread are certain to not escape the current process.

3.4 Programming Model for Shared Heaps

The practicality of our model for direct sharing is a very important aspect of KaffeOS's design. To demonstrate the model's practicality, we implemented a small subset of the Java 1.2 collection API under the shared programming model, and used those data structures to implement a servlet microengine. In this section, we discuss the model's restrictions in detail and show how they affect the implementation of shared objects.

3.4.1 *Restrictions on Shared Heaps.* The design restrictions on shared heaps imply several practical changes to the programming model for shared classes:

Shared Heaps Remain Fixed in Size after Their Creation. All required objects must be allocated while the shared heap is being created. Some collection types, such as `java.util.LinkedList`, use cells that contain a reference to an item in the list and a `next` and `prev` pointer. On a KaffeOS shared heap, we cannot allocate new cells after the heap is frozen. Furthermore, because there is no way to reclaim and reuse memory, we cannot afford to discard removed cells. Instead, we must manage cells manually, which can be done by keeping them on a separate freelist.

Shared Objects Cannot Allocate Data Structures Lazily. All link-time references must be resolved and all bytecode must be translated into native code before freezing a shared heap. This requirement also affects certain programming idioms in the Java code. For instance, the `java.util.HashMap.keySet()` method returns a `Set` object that is created and cached on the first call. Subsequent invocations return the same `Set` object. In our programming model, this object must be created and cached before the shared heap is frozen; for instance, by allocating the `Set` object eagerly in the constructor of `HashMap`.

All Entry Points into the Shared Heaps Must Be Known. The shared heap's size is fixed once the heap is frozen, so no entry items can be allocated afterwards. We introduced an API function that allows the reservation of entry items for specified objects while the heap is created. This function preallocates the entry item; when the write barrier code detects that a reference to the object is written to a user heap, it uses the preallocated entry item instead. Shared classes must be careful to reserve entry items for all objects to which a process might acquire references later.

No References from Shared Heaps to Objects on User Heaps Can Be Created. We must avoid assignments to fields in shared objects, unless the object to which a reference is being assigned is known to also reside on the same shared heap. This requirement precludes the storage of temporary objects, which are allocated on user heaps, in fields of shared objects. Instead, references to such objects must be restricted to local variables on the stack.

These restrictions on our shared programming model result from two independent design goals. The first goal is to avoid sharing attacks and guarantee full memory reclamation. This goal requires the use of write barriers to prevent illegal cross-heap references. The second goal is to prevent programs from asynchronously running out of memory. For this reason, all sharers are charged, which in turn requires that a shared heap is frozen after creation so as to accurately charge all sharers.

3.4.2 *Examples of Shared Data Structures.* We give two examples to detail the implications of our restrictions. As a first example, we created `shared.util.HashMap`, which is an adaptation of `java.util.HashMap` for use on shared heaps. The `shared.util.HashMap` class is an immutable data structure. Map instances are created and populated with (*key*, *value*) pairs during the construction of a shared heap; no elements can be added after the shared heap is frozen.

Kaffe's implementation uses an array of buckets, each of which is an anchor to a singly linked list of `Entry` objects. Each `Entry` object consists of a `next` field, and two fields that hold references to a (*key*, *value*) pair of objects. A hashmap provides methods that return handles to its key and value sets in the form of `Set` objects, which in turn provide iterators. For implementation reuse reasons, the key and value set objects are implemented in an abstract base class `AbstractMap`.

Because the use of iterators on a heap requires direct pointers to the list entries in the hashtable, we must reserve entry items for them. Our implementation is shown in Figure 7. It differs from the original implementation in only three aspects.

(1) It is in the `shared.util` package instead of the `java.util` package. As a side effect, we had to copy and rename `java.util.AbstractMap` to `shared.util.-AbstractMap` as well to provide package access to fields in `AbstractMap`. However, we did not need to duplicate the `Map` interface in the `shared.util` package: hence, shared hashmaps can be used wherever `java.util.Map` instances can be used.
(2) It allocates its `keyset` set and `values` collection eagerly in the constructor.
(3) It reserves entry items for all entries and the key and value collections.

Figure 8 illustrates the interheap connections that are created if an iterator for the set of keys in the hashmap is created. Only three methods and one constructor of `shared.util.HashMap` differ from `java.util.-HashMap`, in that they can throw new errors. For instance, `put()` would throw a `SegmentationViolation` if an attempt is made to modify the hashmap after the shared heap on which it is allocated is frozen. All other methods behave in the same way. In particular, the `entrySet`, `values`, and `keySet` behave as defined in the API specification; they return sets and collections with the same types that a `java.util.HashMap` instance would return.

The shared hashmap implementation provides an example of a immutable data structure. An instance in which the hashmap could be used might be a dictionary that is built once but accessed frequently. If the dictionary changes infrequently, discarding the shared heap that contains the hashmap and copying and updating the hashmap on a newly created shared heap should be viable.

As a second example, we consider the implementation of a service queue in `shared.util.LinkedList`. A service queue is a common communication idiom that is used in client and server applications. The `java.util.LinkedList` class provides a doubly linked list implementation that can be used. However, unlike a hashmap, there are few uses for a queue that do not involve changing the queue's elements. We therefore adapted the implementation to allow for the addition and removal of elements to the queue. Because shared heaps are a fixed size, addition is only possible as long as the total number of added elements

```
package shared.util;
import java.util.Map;

public abstract class AbstractMap
  implements Map
{
  Set keyset;
  ...
  public Set keySet() {
    // create on demand and cache
    if (keyset != null) {
      return keyset;
    }

    keyset = new AbstractSet() {
      ...
      // map key iterator to
      // entry set iterator
      public Iterator iterator() {
        return new Iterator() {
          private Iterator i
            = entrySet().iterator();
          public Object next() { ... }
          ...
        };
      }
    }
    return keyset;
  }
  ...
  public abstract Set entrySet();
}
```

```
public class HashMap
  extends AbstractMap
  implements Map, Cloneable,
             Serializable
{
  public HashMap(...) {
    ...
    Heap h = Heap.getCurrentHeap();
    h.reserveEntryItem(keySet());
  }
  ...
  public Object put(Object key,
                    Object val) {
    Entry e;
    ...
    // Create and add new entry
    e = new Entry(key, val);
    Heap h = Heap.getCurrentHeap();
    h.reserveEntryItem(e);
    e.next = table[bucket];
    table[bucket] = e;
    ...
  }

  public Set entrySet() {
    return
      new AbstractMapEntrySet(this) {
        public Iterator iterator() {
          return new EntryIterator();
        }
        ...
      };
  }

  private class EntryIterator
    implements Iterator
  {
    private Entry next, prev;
    private int bucket;
    public Object next() { ... }
  }
}
```

Fig. 7. Implementation of `shared.util.Hashmap`. By copying the code of `java.util.HashMap` and making slight modifications, we were able to create an implementation of a hashmap that is suitable for use on KaffeOS's shared heaps.

does not exceed the maximum number specified when constructing the `shared.util.LinkedList` object. The actual queue elements themselves must also be preallocated on the shared heap. An example might be a set of preallocated buffers that is read from and written to by the communicating processes.

Figure 9 sketches the object instances involved. We manually manage a number of preallocated `Elem` cells. We changed the original code in `java.util.LinkedList` to add a `freelist` field and methods `getElem` and `putElem` to get

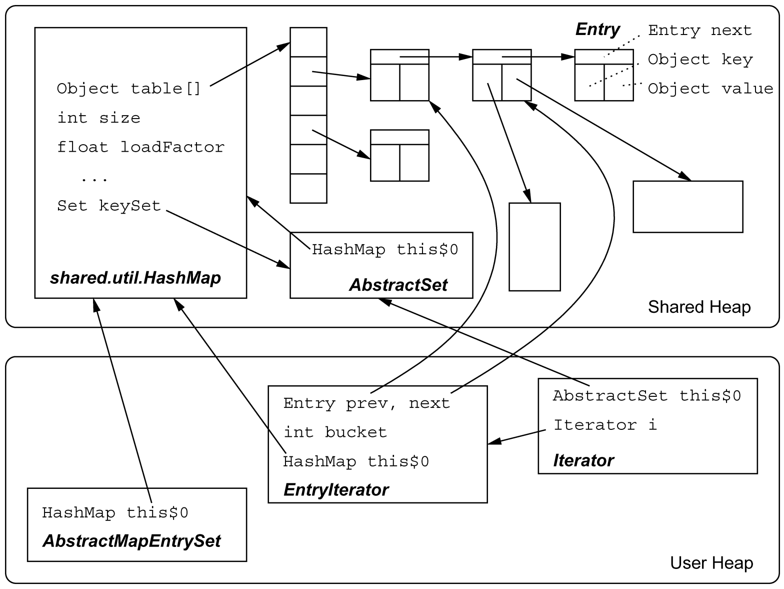


Fig. 8. Use of `shared.util.HashMap` on a shared heap. The `keyset` field in the hashmap object refers to an `AbstractSet` allocated eagerly on the shared heap. Its `iterator()` method returns an anonymous `Iterator` object, which in turn refers to a `HashMap.EntryIterator` object upon which its implementation is based. This entry iterator object is obtained from a call to `HashMap.entrySet()`, which allocates an `AbstractMapEntrySet` object, and a subsequent invocation of the entry set's `iterator` method. In KaffeOS, the use of iterators for hashmaps on a shared heap is unconstrained, because our implementations allocate the key set eagerly and reserve entry items for all `Entry` objects (not shown in the figure).

and put elements on the freelist. We replaced all calls to `new Elem(...)` with calls to `getElem` to get cells from the preallocated list. In addition, we carefully inserted calls `putElem` in those places where a list element is discarded to recycle them. As with shared hashmaps, each list cell requires the reservation of an entry item.

We required some changes to constructors and other methods. A constructor `LinkedList(int)` was created for which the integer argument specifies the maximum number of elements in the list. The `add` and related methods were adapted to throw `SegmentationViolationError` if an object is not on the shared heap. All other methods behave in the same way; in particular, `iterator()` returns an object of type `java.util.Iterator`, and the `shared.util.LinkedList` itself can be used as a `java.util.List` instance.

We used the shared hashmap class and the linked list class to implement a servlet microengine. This engine does not implement the full Java servlet API; however, it is sufficiently complete to demonstrate KaffeOS's interprocess communication. The example includes three processes: a http server process and two servlet engine processes. The server and servlet processes share a queue for http requests on a shared heap. The http server receives http GET requests, examines the URL contained in the request, and dispatches the

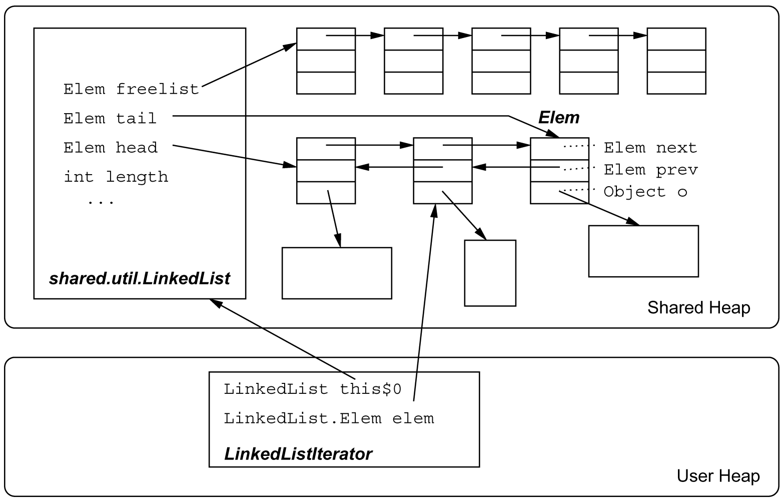


Fig. 9. Use of `shared.util.LinkedList` on a shared heap. Our implementation of `shared.util.-LinkedList`, which can be used for queues, adds a freelist of `Elem` objects to store elements not currently in use. The queue is bounded by the number of `Elem` cells that are preallocated. The use of `LinkedListIterator` instances is unconstrained since we reserve entry items (not shown in figure) for all elements and the `LinkedList` instance itself.

request to a servlet. Each servlet waits for requests on its queue and processes incoming requests. The servlets perform a lookup in a German-English dictionary and display the result to the user. The dictionary is stored in an object of type `shared.util.HashMap` located on a second heap that is shared between the servlet processes.

4. EVALUATION

We focus on three areas in the evaluation of our KaffeOS prototype. First, we measure the run-time overhead KaffeOS introduces, when compared with a JVM that does not provide robust support for multiple processes. We show that the overhead is reasonable (from 0% to 25%) when compared to the Kaffe VM upon which KaffeOS is based. Second, we evaluate KaffeOS's effectiveness in handling untrusted or buggy applications that may engage in denial-of-service attacks. KaffeOS can defend against denial-of-service attacks directed at memory, CPU time, and the garbage collector; KaffeOS's integrity is not compromised by such attacks. Third, we compare KaffeOS to the possible alternative of running multiple applications on multiple JVMs, and show that KaffeOS can outscale that approach.

Our prototype runs as a user-mode application on top of the Linux operating system. All our measurements were taken on a 800 MHz "Katmai" Pentium III, with 256 Mbytes of SDRAM and a 133 MHz PCI bus, running Red Hat Linux 6.2. The processor has a split 32 K Level 1 cache and combined 256 K Level 2 cache. We used the GNU C compiler (Version egcs–1.1.2) to compile the VM, and we used IBM's jikes compiler (Version 1.12) to compile the Java portions of the runtime libraries.

4.1 Application Benchmarks

KaffeOS's use of write barriers for isolation introduces some overhead when running applications. To measure this overhead, we implemented several versions of the system:

No Write Barrier. We execute without a write barrier and run everything on the kernel heap. The *No Write Barrier* version accounts for any possible performance impact of the changes made to Kaffe's runtime and provides the baseline version for examining the write barrier overhead.

No Heap Pointer. This version is the default version of KaffeOS. For each small object page, and for each large object, we store a heap ID in a block descriptor. To avoid cache conflict misses, the block descriptor is not kept at a fixed offset on the same page. Instead, block descriptors are stored in an array, whose index values are computed as affine transformations of the address of an object. At each heap pointer write, the write barrier consists of a call to a routine that finds both the source and the destination object's heap ID from their addresses and performs the barrier checks.

Heap Pointer. In this version, we trade some memory for speed: we increased each object's header by 4 bytes, in which we store the heap ID. In this case, extracting the heap ID is substantially faster, because the block descriptor does not have to be read from memory to retrieve the heap ID. We coded the fast path (i.e., where our policy allows the write to complete) for this barrier version in assembly language.

Fake Heap Pointer. To measure the impact of the 4 bytes of padding in the *Heap Pointer* implementation, we padded the object header by 4 bytes, but did not store the heap ID in them. Instead, we determine the heap ID as in the *No Heap Pointer* version. In other words, we impose the memory overhead of the *Heap Pointer* version, but do not use a faster write barrier.

The KaffeOS JIT compiler does not inline the write barrier routine. Inlining the write barrier code might improve performance, but it could also lead to substantial code expansion. For instance, in the *No Heap Pointer* version, the write barrier code includes 29 instructions, plus an additional 16 instructions for procedure calling conventions, which an inlined version may not require. Ideally, the intermediate representation of the write barrier code should be available to a just-in-time compiler, so that the compiler can apply heuristics to decide whether to inline or outline on a per-call site basis.

4.1.1 *Write Barrier Overhead.* We used the SPEC JVM98 benchmarks [SPEC 1998] to evaluate the performance of KaffeOS for real-world, medium-sized applications. These benchmarks measure the efficiency of the just-in-time compiler, runtime system, operating system, and hardware platform combined. SPEC JVM98 consists of 7 benchmarks: compress, jess, db, javac, mpegaudio, mtrt, and jack. All except db are real-life applications that were developed for purposes other than benchmarking. An eighth benchmark, check, is used to test the proper implementation of Java's features in the VM; KaffeOS passes the check benchmark.

Table II. Number of Write Barriers Executed by SPEC JVM98 Benchmarks

		Cross-heap	
Benchmark	Barriers	Number	Percent
compress	47,969	3336	6.95%
jess	7,939,746	4747	0.06%
db	30,089,183	3601	0.01%
javac	20,777,544	5793	0.03%
mpegaudio	5,514,741	3575	0.06%
mtrt	3,065,292	3746	0.12%
jack	19,905,318	6781	0.03%

The "cross-heap" column counts the occurrences where source and destination do not lie in the same heap; that is, those for which entry and exit items must be created or updated.

compress compresses and decompresses data from a set of five tar files using a modified Lempel-Ziv method (LZW). Compress does not use many objects; it spends most of its execution operating on two large byte arrays. **jess** is a Java Expert Shell System (JESS) based on NASA's CLIPS expert shell system. This benchmark is computationally intensive and allocates many short-lived objects. **db** is a synthetic benchmark that simulates database operations on a memory-resident database. **javac** is the Java compiler from Sun Microsystem's Java Development Kit (JDK) version 1.0.2. Its workload is the compilation of the jess benchmark. **mpegcompress** decompresses audio files that conform to the ISO MPEG Layer-3 audio specification. This benchmark performs very little memory allocation. **mtrt** is a raytracer program that renders a small scene depicting a dinosaur. It is the only multithreaded benchmark. The **jack** parser generator is based on the compiler toolkit now known as JavaCC. The workload consists of a file that contains instructions for the generation of jack itself.

In addition to the four different write-barrier implementations mentioned above, we include in our comparison the version of Kaffe on which KaffeOS is based. We use a development snapshot from June 2000 for that purpose; we label this version "Kaffe 2000" in our benchmarks. We also include IBM's JVM [Suganuma et al. 2000] from the IBM JDK 1.1.8, which provides one of the fastest commercial JIT compilers available for JVMs that implement Java version 1.1.x. Our prototype does not provide support for the abstract windowing toolkit (AWT), which prevents us from running the benchmark according to the SPEC JVM98 run rules. Our results are not comparable with any published SPEC JVM98 metrics.

We instrumented KaffeOS to determine the number of write barriers that are executed in a single run of each benchmark. Except in the *No Write Barrier* case, we run each benchmark in its own process. Table II shows how many write barriers are executed for a single run. The number of write barriers depends not only on the application, but also on implementation decisions in the runtime libraries used. The table shows that the source and destination objects lie in the same heap for almost all write barrier executions counted. Only a miniscule fraction of write barriers (less than 0.2%, except for compress, which

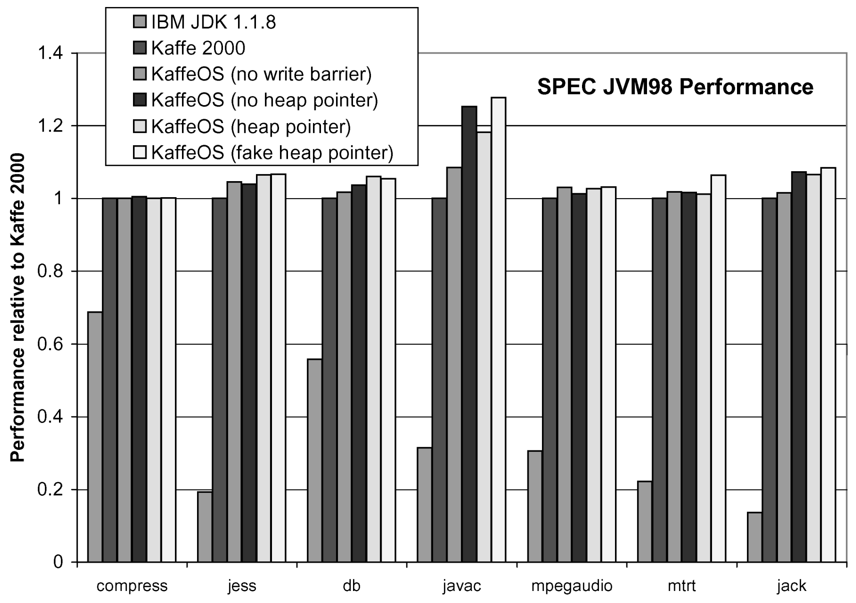


Fig. 10. KaffeOS performance in SPECjvm98 benchmarks. This chart displays the time needed to run SPECjvm98 using the provided harness, relative to the time needed by Kaffe 2000. Kaffe 2000 is the version of the Kaffe VM upon which KaffeOS is based. All Kaffe-based versions are roughly 2–8 slower than IBM's industrial VM. The differences between Kaffe 2000 and KaffeOS are small, and the overheads of all write barrier versions with respect to the *No Write Barrier* version are tolerable.

does hardly any writes) are "cross-heap," in that they result in the creation or update of entry and exit items. The other possible outcome produced by a write barrier, a segmentation violation error, should not and does not occur in these benchmarks. Differences in the number of "cross-heap" write barriers result when applications use different kernel services. For instance, javac opens many more files than compress.

Figure 10 shows the results of running the SPECjvm98 benchmarks. We ran each benchmark three times in the test harness provided by SPEC (in a run that follows all of SPEC run rules, each benchmark is run two to four times). We set the maximum heap size to 64 MB for each JVM. The bars show the average runtime as displayed by the test harness, normalized with respect to Kaffe 2000.

Overall, IBM's JVM is between 2–8 times faster than Kaffe 2000. We focus on the differences between Kaffe 2000 and the different versions of KaffeOS. For those benchmarks that create little garbage, compress and mpegaudio, the difference in total runtime is small. For the other benchmarks, we observe a larger difference, even between Kaffe 2000 and the *No Write Barrier* version of KaffeOS. This difference exists because of the changes we made to the runtime system, but also because the time spent in garbage collection differs from benchmark to benchmark.

Table III. Measured vs. Best-Case Overhead

	Total Runtime		Runtime Excluding GC Time	
Benchmark	*No Heap Ptr*	*Heap Ptr*	*No Heap Ptr*	*Heap Ptr*
compress*	−0.5%(0.0%)	0.0%(0.0%)	0.1%(0.0%)	0.5%(0.0%)
jess	−1.1%(1.0%)	1.7%(0.3%)	−0.5%(1.2%)	2.3%(0.3%)
db	2.5%(3.8%)	4.5%(1.1%)	3.5%(4.1%)	4.8%(1.2%)
javac	5.6%(2.5%)	3.2%(0.7%)	7.7%(3.4%)	2.9%(0.9%)
mpegaudio*	−1.4%(0.8%)	−0.7%(0.2%)	−1.2%(0.8%)	−0.5%(0.2%)
mtrt*	0.3%(0.5%)	−0.4%(0.1%)	3.5%(0.6%)	−0.1%(0.2%)
jack	3.4%(1.9%)	2.8%(0.5%)	4.2%(2.4%)	2.4%(0.6%)

*small number of write barriers.
In each table cell, the first number is the measured overhead; the number in parentheses is the best-case overhead relative to the *No Write Barrier* version.

Table III compares the measured overhead to the overhead that could be expected from the write barriers alone, assuming the best-case cycle counts. The worst overhead measured is 5.6% with respect to runtime excluding GC time; 7.7% with respect to the total runtime without the heap pointer optimization; and 4.5% and 4.8%, respectively, if the optimization is applied. For two benchmarks that perform a substantial number of writes, java and jack, the actual penalty is predictably larger than the estimate obtained using a hot cache. For these two benchmarks, the *heap pointer* optimization is effective in reducing the write barrier penalty. Excluding GC, KaffeOS *Fake Heap Pointer* performs similarly to KaffeOS *No Heap Pointer*; however, its overall performance is lower because more time is spent during GC.

There are some anomalies: jess runs faster with write barriers than without, and the overhead of db is lower than the expected best-case overhead. The relative performance order is not consistent. Without nonintrusive profiling, which we do not have available for Kaffe, we can only speculate as to what the reasons might be. It is possible that cache effects are to blame, since both versions have completely different memory allocation patterns. Some internal data structures, such as the table of interned strings, are instantiated only once in the *No Write Barrier* cases, whereas they are duplicated for each heap in all other cases. Finally, since the garbage collector collects only the user heap, we may see a small generational effect, since fewer objects need to be walked during each process's GC.

On a better system with a more efficient JIT, the relative cost of using write barriers could increase. However, a good JIT compiler could perform several optimizations to remove write barriers. Static analysis could be used to remove redundant write barriers, as demonstrated by Zee and Rinard [2002]. To perform such optimizations, the compiler would have to be extended to know about how KaffeOS uses write barriers, so that it could infer when write barriers are redundant. If a generational collector were used, we should be able to combine the write barrier code for cross-heap checking with the write barrier code for cross-generation checking.

4.1.2 *Overhead for Thread Stack Scanning.* Each KaffeOS process must scan all threads when garbage collecting its heap. Our experiments show that this requirement does not lead to priority inversion between processes. It can, however, impose significant overhead for each process, which we quantify for our prototype.

Priority inversion could occur if one process prevented a second process from running because it needed to stop the second process's threads to scan them. We stop remote threads on three occasions during the mark phase of a heap's garbage collection: first, when marking entry items, second, when scanning an individual remote thread's stacks, and third, when recoloring and rewalking the list of black items. The potential for priority inversion depends on the maximum amount of time remote threads must be stopped, which is the maximum of the amounts of time spent on marking and recoloring entry items, respectively, combined with the maximum amount of time spent on scanning any individual remote thread stack. The overall maximum does not depend on the number of remote threads.

All of the aforementioned times are application-specific. To estimate how much time would be spent marking and recoloring, we ran the SPEC JVM98 in parallel with a process that was instructed to repeatedly garbage collect its own heap. We found that a very small amount of time was spent in theses phases (about 4.7% and 3.6% of a 10 ms time slice on a 800-MHz machine, or 383 K and 291 K cycles.)

To estimate how much time is spent scanning a remote thread stack, we used a scenario in which the effective stack size was varied using a recursive function with varying levels of recursion depth. Our experiments showed that the time spent scanning increased linearly with the stack size, as could be expected. For the maximum size in our experiment, a thread stack of length 221,000 bytes, thread stack scanning took about 39% of a time slice or 3.1 M cycles, which corresponds to a delay of 4.8 ms. Typical stack sizes are much smaller: in the vast majority of cases, fewer than four pages per thread must be scanned.

We also estimated the total overhead introduced by remote thread scanning. Based on the SPEC benchmarks, we estimated for n threads with stack sizes $s_i, i = 1..n$ an overhead of roughly $\sum_{i=1}^{n} 28 * s_i$ cycles. As an example, for 100 remote thread stacks of 8 kilobytes average size, we could expect about 23 * 10^6 cycles or about 29 ms per garbage collection. This overhead is substantial, which is why limits on the number of threads must be applied.

Several optimizations could reduce the $O(mn)$ complexity for m processes and n threads exhibited by our current implementation. The information from a stack scan by one process's garbage collector could be saved and made available to all collectors (a time-space trade-off). Incremental thread stack scanning [Cheng et al. 1998] could be used to reduce the time required to scan a stack. Finally, we could avoid scanning a stack multiple times while it is suspended.

4.2 Denial-of-Service Scenarios

We evaluate KaffeOS's ability to prevent denial-of-service attacks using a Java servlet engine. A Java servlet engine provides an environment for running multiple Java servlets at a server. We used the off-the-shelf Apache 1.3.12 web server, the JServ 1.1 servlet engine [Java Apache Project 2000], and Sun's JSDK 2.0 release of the Java servlet extensions. We modified a version of the Apache benchmark program *ab* to simulate multiple clients on a single machine. Each client issues HTTP GET requests for a specified URL on the server. On the server machine, the Apache server processes and forwards these requests to one or more JServ engines. An Apache module (*mod_jserv*) communicates with the servlet engines using a custom protocol called ajpv12.

Each JServ instance can host one or more *servlet zones*, which are virtual servers. Although a servlet zone can host multiple servlets, in our experiment each servlet zone hosts exactly one servlet for simplicity. A URL is mapped to each servlet; the servlet is loaded when a user issues the first request for its corresponding URL. Servlets are kept in memory for subsequent requests; if no requests arrive for a certain time period, they are unloaded and garbage collected.

We compared three configurations:

IBM/n: We run multiple servlets in one servlet engine; the servlet engine runs in a single instance of the IBM JVM.

KaffeOS: We run each servlet in a separate engine; each servlet engine runs in its own process on top of KaffeOS.

IBM/1: We run each servlet in a separate engine; each engine runs in a separate instance of the IBM JVM in a separate process on top of the Linux operating system.

We consider three scenarios that involve different kinds of denial-of-service attacks. First, we attempt to deny memory to other servlets by running a servlet that allocates large amounts of memory until it exceeds its limit and is terminated. We also use this scenario to verify that KaffeOS can terminate such applications safely. Finally, we examine attacks against CPU time and the garbage collector.

4.2.1 *MemHog Servlet.* In this scenario, we use small "Hello, World" servlets as examples of well-behaved servlets that provide a useful service. Alongside these well-behaved servlets, we run a "MemHog" servlet that attempts to deny memory to them. This servlet, when activated, spawns a thread that enters a loop in which it repeatedly allocates objects. The objects are kept alive by linking them in a singly linked list. In the IBM/1 and IBM/n configurations, allocations will fail once the JVM's underlying OS process reaches its maximum heap size. In the KaffeOS configuration, allocations will fail when the process reaches its set limit. In all three configurations, an `OutOfMemory` exception is eventually thrown in the thread that attempts the first allocation that cannot succeed.

The `OutOfMemory` exception can occur at seemingly random places, and not only in the thread spawned by the MemHog. In the IBM/n scenario, a thread can run out of memory in the code that manipulates data structures that are shared between servlets in the surrounding JServ environment; in all three scenarios, a thread can run out of memory in code that manipulates data structures that are kept over successive activations of one servlet. Eventually, these data structures become corrupted, which results in an unhandled

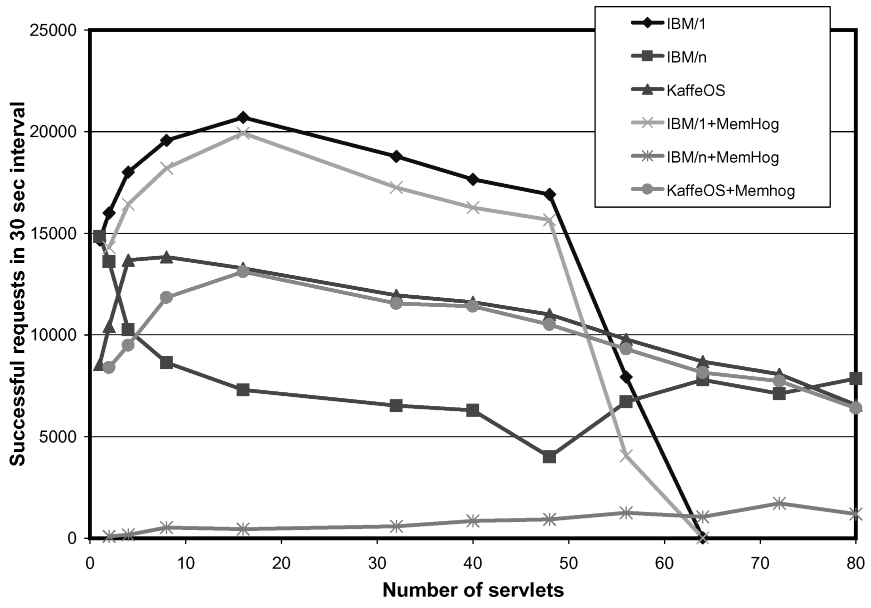


Fig. 11. Throughput for MemHog. This chart displays the number of successful requests received in 30 seconds. *IBM/1* is successful at isolating the MemHog, while *IBM/n* is not. *IBM/n* and KaffeOS can support more than 64 servlets, while *IBM/1* cannot.

exception in one or more threads. In KaffeOS, this event causes the underlying KaffeOS process to terminate. In the IBM/1 and IBM/n scenarios, the underlying JVM terminates. In some instances, we observed the IBM JVM crash, which resulted in segmentation violations at the OS process level. When simulating this denial-of-service attack, we did what a system administrator concerned with availability would do: we automatically restarted the JVMs and the KaffeOS processes, respectively, whenever they crashed because of the effects caused by a MemHog.

We counted the number of successful responses our clients received from the "Hello, World" servlets during a certain period of time (30 seconds). We consider this number an effective measure for the amount of service the system provides, because it accounts for the effects caused by the denial-of-service attack on the Apache/JServ system as a whole. For each configuration, we measured the amount of service provided with and without a denial-of-service attack. We ran the experiment with a total of 2, 4, 8, 16, 32, 40, 48, 56, 64, 72, and 80 servlets. We allowed up to 16 concurrent connections from the client; increasing that number further did not increase throughput. The client rotated these connections among all servlets, so that all servlets were activated shortly after the start of the experiment and remained active throughout the experiment.

Figure 11 shows the results. This figure shows that the KaffeOS configuration, as well as the IBM/1 configuration, can successfully defend against this denial-of-service attack, because the impact of the single MemHog is isolated. However, the graph shows that running each of the servlets in a single JVM, as done in the IBM/1 approach, does not scale. We estimate that each

IBM JVM process takes about 2MB of virtual memory upon startup. Starting multiple JVMs eventually causes the machine to thrash. An attempt to execute 100 IBM JVMs running the Apache/JServ engine rendered the machine inoperable.

The IBM/n configuration, on the other hand, can easily support 80 servlets. However, if the MemHog is added, this configuration exhibits a severe decrease in performance. This degradation is caused by a lack of isolation between servlets. As the ratio of well-behaved servlets to malicious servlets increases, the scheduler yields less often to the malicious servlet. Consequently, the service of the IBM/n,MemHog configuration shown in Figure 11 improves as the number of servlets increases. This effect is an artifact of our experimental setup and cannot be reasonably used to defend against denial-of-service attacks.

In addition to being able to defend against the denial-of-service attack, KaffeOS can support as many servlets as IBM/n, but its performance does not scale very well. We identified two likely sources for the behavior. The first source are ineffiencies in the signal-based I/O mechanisms used in our prototype's user-level threading system. The second source are the deficiencies in KaffeOS's memory allocation subsystem, which we discussed in Section 3.3. Specifically, we found that increased external fragmentation slightly increased the garbage collection frequency of individual processes, because it became harder for them to expand their heap even though they had not reached their limits. As discussed earlier, a moving collector should be able to alleviate this limitation.

We conclude from these experiments that KaffeOS's approach of supporting multiple applications in a single JVM can effectively thwart denial-of-service attacks directed against memory. The operating system-based approach (IBM/1) has the same ability, but its scalability is restricted when compared to KaffeOS.

4.2.2 *MemHog Stress Test.* To demonstrate that KaffeOS reclaims all of a process' memory upon termination, we developed a stress test in which we repeatedly activate a single MemHog, and kill it once it exceeds its memory limit. After each kill of a MemHog, we record the overall number of bytes allocated by all heaps, as well as the number of bytes in objects on the kernel heap. We found that both byte numbers remained almost constant over time—linear regression analysis revealed a leakage of 31.5 bytes per kill, which indicates that KaffeOS can reclaim virtually all of the memory used by the MemHog.

To demonstrate that KaffeOS is robust, we ran this test with a debugging version of KaffeOS that includes a series of internal integrity checks for this test. If any inconsistency is detected during one of these tests, we abort the JVM. We cannot claim that our prototype has the robustness and maturity of an industrial product. MemHog will crash it at some point (between 1000 and 5000 kills) because of bugs in our prototype. However, when compared to *IBM/n*, which typically survives only a few MemHogs throwing `OutOfMemory` exceptions, this test provides strong evidence that the introduction of a user/kernel boundary is important for the construction of a system that is robust enough to safely terminate ill-behaved applications.

4.2.3 *CpuHog Servlet.* A denial-of-service attack against CPU time is harder to detect than an attack against memory, because it can be difficult to determine whether a servlet's use of the CPU provides a useful service or is merely wasting CPU time. We do not attempt to provide a general solution to this problem. Instead, we show how our CPU management mechanisms can be used to ensure that a servlet's clients obtain the share of service that is provisioned for them, even in situations in which another servlet attempts to use more than its share.

We adopt the view that using the CPU when it would otherwise be idle is harmless. In systems that bill users for CPU time used, this view would not be appropriate. We consider a successful denial-of-service attack against the resource CPU a situation in which an important task cannot get done or is being done much slower, because the CPU is used by some other, less important task. Our scheduling scheme cannot limit the amount of CPU time a process consumes: it can only guarantee a certain amount to processes that are runnable.

We used an MD5-Servlet as an example of a servlet that performs a computationally intensive task. MD5 is a one-way hash function that is used to digitally sign documents [Rivest 1992]; hence, this servlet could be seen as representative of an application that provides notary services over the World Wide Web. For each request, the servlet computes the MD5 hash function over the first 50,000 words of the `/usr/dict/word` dictionary file. As a measure of service, we use the number of successfully completed requests per second. Our modified version of *ab* estimates this number of requests per second simply by inverting the time difference between consecutive responses. We also ran a "CPUHog" servlet that executes an infinite loop in which it does not voluntarily yield the CPU to other threads.

For the purposes of this experiment, we evaluated a possible scenario in which a hog is activated and in which CPU shares must be adjusted so as to thwart a possible denial-of-service attack. Four servlet zones start out with equal shares of 1/4 each. The fourth servlet zone has no servlet running, so we expect the three servlets to split the CPU time among them. After a few seconds have passed, we activate the CPUHog. Once the CPUHog is activated, it immediately uses its full share: it is limited to a 1/4 share of the CPU. We assume that some external mechanism or watcher detects that this CPU consumption by the Hog is undesirable. We then set the share of the Hog to zero, after which the three MD5 servlets should again split the CPU evenly. Finally, we assume that servlet C's throughput should be increased. Servlet C's share is increased to 1/2, and A and B's shares are reduced to 1/4, after which we expect to see an identical ratio in the observed throughput of these three servlets.

Figure 12 shows our results for the CPUHog. When activated, the CPUHog simply sits in a loop and consumes CPU cycles. The graph is stacked, that is, the area between the lines indicates what share each servlet received. The straight line at the top was determined by measuring the aggregate average throughput of the MD5 servlets with no Hog. It provides an approximation of the maximum throughput capacity in our system. The CpuHog's area is computed as the difference between this expected maximum capacity and the sum of the throughputs of the A, B, and C servlets.

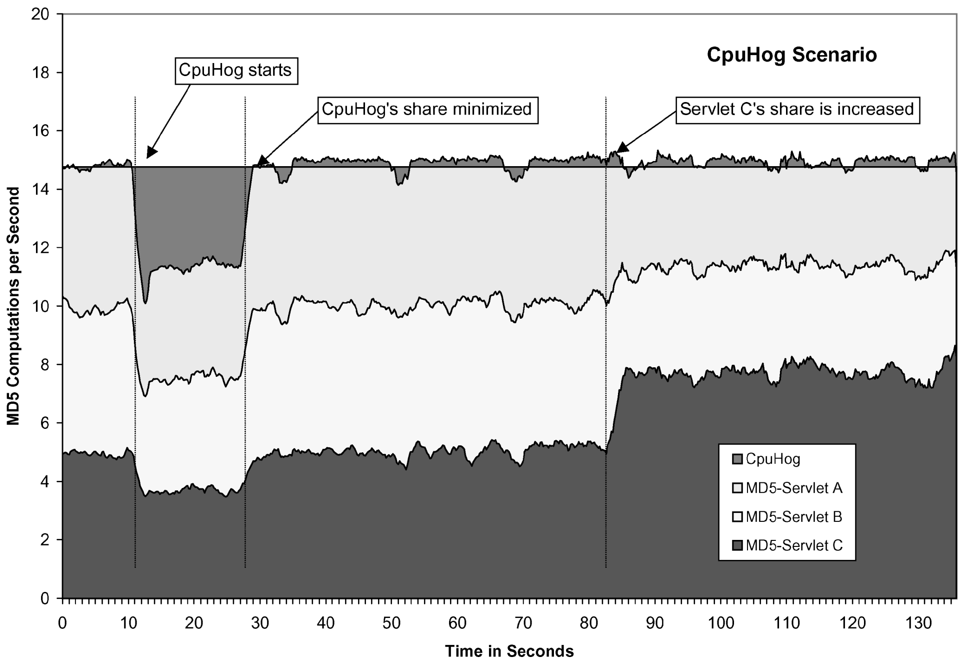


Fig. 12. Throughput for CpuHog scenario. The measured throughput matches the expected throughput in this scenario.

For implementation reasons, we give each task (even those with no share) at least one ticket in the stride scheduler. This implementation artifact accounts for the slight periodic dents at the top, in which the CPUHog gets to run for a brief period of time, despite having been assigned a zero share. We disallow zero tickets to avoid having to provide for CPU time to execute the exit code for applications whose share is reduced to zero—other policies for handling this situation are possible.

4.2.4 *GarbageHog Servlet.* In the last scenario, we replaced the CPUHog with a "GarbageHog" servlet that attempts to hog the garbage collector. The GarbageHog produces some garbage at each invocation; specifically, it allocates 25,000 objects of type `Integer`. Unlike the MemHog, it does not keep those objects alive, but drops all references to them. The garbage collector has to collect these objects to reclaim their memory. Unlike the CpuHog, the GarbageHog serves requests: in this respect, it resembles a well-behaved servlet, except for its generation of excessive and unnecessary garbage.

The GarbageHog results are similar to the CpuHog, which indicates that KaffeOS is successful at isolating the garbage collection activity of this servlet. In other words, it demonstrates that we were successful in separating the garbage collection activities of different processes. The garbage collector is no longer a resource that is shared between processes, but has become an activity within a process that is subject to the stride scheduler's scheduling policy, along with all other activities in the process. Note that our results do not measure how accurately our prototype's scheduler assigns CPU time.

5. RELATED WORK

A great deal of the work on Java grew out of work on single-language operating systems such as Pilot [Redell et al. 1980] and Cedar [Swinehart et al. 1986]. Other more recent single-language systems include the Juice system [Franz 1997] for Oberon [Wirth and Gutknecht 1992], the SPIN kernel [Bershad et al. 1995b], Inferno [Dorward et al. 1997], and the .NET system from Microsoft Corporation [2003]. Similar systems include single-address-space operating systems such as Opal [Chase 1995], Nemesis [Roscoe 1995], and Angel [Wilkinson et al. 1992], as well as the transaction-based VINO system [Seltzer et al. 1996]. None of these systems provided the combination of language-based protection and resource management that KaffeOS provides.

In this section, we concentrate on relevant work on Java. Several other researchers and developers have built systems for Java applications that address some of the problems KaffeOS is designed to address. Early systems [Balfanz and Gong 1998; Bernadat et al. 1998; Saulpaugh and Mirho 1999] provided simple multiprocessing facilities for Java without addressing isolation and resource management. Other systems focused on the increased scalability that can be obtained by running multiple applications in one JVM [Czajkowski 2000; Dillenberger et al. 2000; Lizt 1999], but they did not address resource control. Various other systems also provide process models for Java but use different approaches, make different assumptions about the environment in which they run, or use different sharing models [Hawblitzel et al. 1998; Tullmann 1999; van Doorn 2000]. We discuss in more detail several of these systems; each of them addresses only a single issue, such as resource control [Czajkowski and von Eicken 1998], memory management [Bollella et al. 2000], or termination [Rudys and Wallach 2002]. Our work, on the other hand, addresses these issues in a unified framework.

The MVM (Multitasking Virtual Machine) [Czajkowski and Daynés 2001] is a JVM developed at Sun Labs that can support multiple processes. MVM's implementation can share text between processes, so achieves better efficiency than KaffeOS. It achieves isolation by replicating as much state as possible. Finally, it allows untrusted native code to be executed safely by executing such code within its own OS-level process.

JSR-121 [Java Community Process 2003] is a proposed addition to Java that has passed public review. It provides the notion of an *isolate*, which is a lightweight process. Although it does not have all of the features of a KaffeOS process, it is the first step in providing such functionality in the Java standard. It illustrates the fact that the Java community has recognized the need for supporting application isolation.

J-Kernel, JRes, and Luna. The J-Kernel system by Hawblitzel et al. [1998] is a layer on top of a standard JVM that adds some operating system functionality. Its microkernel supports multiple protection domains called tasks. Communication in the J-Kernel is based on capabilities. Java objects can be shared indirectly by passing a pointer to a *capability* object through a "local RMI" call. The capability is a trusted object that contains a direct pointer to the shared object. Because of the level of indirection through capabilities to shared objects, access to shared objects can be revoked. As with shared classes in KaffeOS, a capability can be passed only if two tasks share the same class through a common class loader.

All arguments to intertask invocations must either be capabilities, or must be copied completely. By default, standard Java object serialization is used, which involves marshaling into and unmarshaling from a linear byte buffer. To decrease the cost of copying, a fast copy mechanism is also provided. In contrast, KaffeOS provides its direct sharing model as a means of interprocess communication. Although not optimized for indirect sharing, KaffeOS does not preclude the use of object serialization. In addition, in KaffeOS a task's primary access to system services such as filesystem or network access is direct through the KaffeOS kernel.

The J-Kernel's use of indirect sharing, combined with revocation of capabilities, allows for full reclamation of a task's objects when that task terminates. However, because the J-Kernel is merely a layer on top of a standard JVM, it cannot isolate the garbage collection activities of different tasks, nor can it account for other shared functionality.

The J-Kernel supports thread migration between tasks if a thread invokes a method on an indirectly shared object. The thread logically changes protection domains during the call; a full context switch is not required. To prevent malicious callers from damaging a callee's data structures, each task is allowed to stop a thread only when the thread is executing code in its own process. This choice of system structure requires that a caller trust all of its callees, because a malicious or erroneous callee might never return. In contrast, KaffeOS can always terminate uncooperative threads safely with respect to the system, but interprocess communication will involve at least one context switch.

JRes [Czajkowski and von Eicken 1998], a companion project to the J-Kernel, provides a way to control and manage computational resources of Java applications. Like the J-Kernel, JRes can run on top of a standard JVM. It controls CPU time, memory, and network bandwidth. CPU time is accounted for by using hooks into the underlying OS, which is responsible for scheduling them. JRes lowers the priority of threads that have exceeded their resource limit. Memory is accounted for by rewriting Java bytecode in a way that adds bookkeeping code to instructions that allocate memory. JRes adds finalization code to classes so that tasks can be reimbursed for allocated objects.

Luna [Hawblitzel and von Eicken 2002] is a system that extends the Java language to support intertask communication via special types. It is designed to enable arbitrary sharing between processes while still preserving safe termination. Special types are used for interprocess object references; these references all go through an extra indirection at runtime to determine their validity. As a result, when process termination occurs, remote references can be invalidated with a single write.

Alta. The Alta [Tullmann and Lepreau 1998; Tullmann 1999] system provides an environment that runs multiple applications in a single JVM that is modeled after the Fluke microkernel [Ford et al. 1996]. The Fluke microkernel provides a nested process model, in which a parent process controls all aspects of its child process. It does so by interposing on the child's communication, which is done via IPC. As in many microkernels, system services are provided through servers, which complicates per-task accounting.

In Alta, capabilities provide a way to share objects between processes indirectly, but there is no mechanism to prevent the leaking of references to objects in one process to another. A parent task is responsible for restricting a child's communication so object references are not leaked. As a result, Alta cannot always guarantee full reclamation. Objects can be directly shared between processes if their types, and the closure of the types of their fields, are structurally identical. This approach constitutes an extension of Java's type system. In contrast, KaffeOS guarantees the type safety of its shared objects through the use of a common class loader.

Alta provides the ability to control a process's memory usage, which is also done by the process's parent. Unlike KaffeOS, Alta does not separate the memory resource hierarchy from the process hierarchy. Because Alta does not provide separate heaps, it cannot account for garbage collection activity on a per-process basis.

Secure JVM on Paramecium. A secure JVM that runs on top of the Paramecium extensible operating system [van Doorn et al. 1995] is described in van Doorn [2000]. Van Doorn argues that language-based protection is insufficient and prone to bugs. The multiple protection domains his JVM supports are separated using hardware mechanisms, which provide strong memory protection and eliminate the need to trust the bytecode verifier.

In the Paramecium JVM, each class or instance belongs to a domain. The language protection attributes of a class or instance, such as private or protected, are mapped to page protection attributes. As a result, an object's fields may have to be split over multiple pages. Classes and objects are shared between domains if they are placed in the same page. False sharing could occur if a page contains both a shared object and another, unrelated object that should not be shared. Because such false sharing would violate the confinement of domains, the garbage collector moves objects or parts of objects between pages to prevent this situation. Conversely, the garbage collector attempts to move objects that can be safely shared to pages with the same set of access permissions.

Garbage collection costs are not separately accounted for. In addition, in order to not undermine the gain in safety added by using hardware protection, the collector must be written in a way that can handle possible corruption of the objects it examines. This necessity leads to increased overhead and can lead to tricky code.

Real-Time Java (RTJ). Real-Time Java (RTJ) is a standard for programming embedded devices. Unlike KaffeOS, RTJ is not concerned with untrusted or buggy applications but instead focuses on providing bounds on the execution time of applications. In particular, RTJ provides facilities to manage memory in a way that allows for real-time execution bounds in the presence of automatically managed memory. Despite dissimilar goals, RTJ and KaffeOS use similar mechanisms.

Threads in RTJ can allocate their objects from *scoped memory*. Such memory areas are not garbage collected and can be freed once all threads have finished using them. Scopes are similar to explicit regions [Gay and Aiken 1998], which are a memory management scheme in which a programmer can allocate objects in different regions but free regions only as a whole. As in KaffeOS, write barriers are used to ensure this property at run time; a runtime error is triggered when an attempt to establish a reference from an outer, longer-lived scope into an inner, short-lived scope is being made. Unlike KaffeOS heaps, scoped memory is not garbage collected and does not allow any cross-scope references.

RTJ also provides a mechanism to asynchronously terminate threads. To address the problem of corrupted data structures, RTJ allows the programmer to declare which sections of code are safe from termination requests and in which sections of code such requests must be deferred. This use of deferred termination is similar to the KaffeOS user/kernel boundary. Unlike in KaffeOS, entering a region in which termination is deferred is not a privileged operation. The goal of terminating threads asynchronously in RTJ is not to be able to kill uncooperative applications but to provide the programmer with a means to react to outside events.

RTJ's designers acknowledge that writing code that is safe in the face of asynchronous termination is hard. For this reason, asynchronous events are deferred by default in legacy code, as well as during synchronized blocks and constructors. Only code that explicitly enables asynchronous events by using a throws clause can be asynchronously terminated. KaffeOS's user/kernel boundary does not provide termination as a means of programming multithreaded applications, but solely as a means to terminate threads as a process exits.

Soft Termination. Rudys and Wallach [2002] propose a scheme for safe termination of "codelets" in language-based systems. Their work focuses only on termination and does not include other aspects of a fully developed process model, such as memory control or sharing. In their scheme, a codelet's class file is rewritten such that a "check-for-termination" flag is added to each class. The bytecode is rewritten to check this flag whenever a non-leaf method is invoked and on every backward branch. System code is not rewritten in this way. Hence, as in KaffeOS, termination is deferred while a thread is executing system code. The authors give a formal semantics for the rewrite rules and prove that a thread terminates in a bounded amount of time.

Rudys and Wallach argue that a user/kernel boundary is too inflexible for a language-based system because a thread can cross in and out of system code. Although user and system code can indeed both call each other, it does not follow that calls in both directions can be treated alike. In their system, just as in KaffeOS, every call from system code to user code must be treated as an upcall.

The semantics and expressiveness of the soft termination scheme appears mostly identical to KaffeOS's user/kernel boundary. Their implementation has the important advantage that it functions on a standard Java VM, but the price they pay is a 3–25% overhead for checking the flag, even if no termination request is pending (the common case). In comparison, KaffeOS only performs such a check when leaving kernel mode.

JanosVM. JanosVM [Tullmann et al. 2001] is a VM for active networks that uses KaffeOS as its design and implementation base. Unlike KaffeOS, JanosVM is not intended as an environment in which to run applications directly: it is intended as a foundation on which to build execution environments for active code. JanosVM both restricts KaffeOS's flexibility by specialization for active code, and extends its flexibility by providing trusted access to some kernel functionality.

JanosVM supports direct cross-heap references, but does not use entry and exit items the way KaffeOS does. The responsibility of managing entry and exit items falls on the designer who uses JanosVM to build an execution environment. An advantage of explicitly managed cross-heap references is that heaps can be reclaimed immediately after a process terminates, as is the case in the J-Kernel.

KaffeOS's kernel boundary is static, but it also exports primitives for entering and leaving kernel mode to trusted parties. JanosVM makes use of this facility in a systematic way for interprocess communication between trusted parties. This extension protects trusted servers without making them statically part of the kernel, and can be viewed as a microkernel approach to structuring a Java runtime.

Because JanosVM is not intended to support general-purpose applications, shared heaps are not supported. Instead, JanosVM supports customized sharing for its application domain. For instance, processes can share packet buffers, which are objects with a well-known set of access primitives that are treated specially by the system. Consequently, some implementation complexity was removed, because the kernel garbage collector does not need to check for orphaned shared heaps.

JanosVM improves on KaffeOS's resource framework by including other resources in a uniform way. For instance, the namespace of a process is treated as a resource that can be managed just like the memory and CPU time used by a process. In KaffeOS, such policies would require a new process loader implementation.

6. CONCLUSION

We have presented the design and an implementation of a Java runtime system that supports multiple, untrusted applications in a robust and efficient manner. We have used operating system ideas to introduce a process concept into the Java virtual machine. KaffeOS's processes provide the same properties as operating system processes: they protect and isolate applications from each other, and they manage and control the resources that they use.

We have demonstrated that separating "kernel" code from "user" code is necessary to protect critical parts of the system from corruption when applications are killed. Code that executes in kernel mode delays termination requests until it returns to user mode; in addition, it is written such that it can safely back out of exceptional situations. This separation does not prevent sharing, because it does not change the way in which objects access each other's fields and methods; type safety remains the means to enforce memory safety.

To ensure full reclamation of a process's memory, we depend on proper kernel design and provide each process with a separate heap. To ensure that heaps are separate, KaffeOS uses write barriers, which are a garbage collection technique. This use of write barriers prevents sharing attacks, in which a process could prevent objects from being reclaimed by collaborating with or duping another process into holding onto references to these objects.

KaffeOS adopts distributed garbage collection techniques to allow for the independent collection of each process's heap, even in the face of legal cross-references between the kernel heap and user heaps. The key idea is to keep track of the references that point into and out of each user heap. Incoming references are kept track of in entry items, which are treated as garbage collection roots. Outgoing references are kept track of in exit items, which allow for the reclamation of foreign entry items when their reference counts reach zero. These ideas were originally developed for situations where objects cannot access other objects directly. KaffeOS does not prevent direct access, but uses entry and exit items for resource control.

In addition to kernel-level sharing for increased efficiency, KaffeOS also supports user-level sharing through shared heaps. Shared heaps allow processes to share objects directly, which allows for efficient interprocess communication in KaffeOS. However, to not compromise full reclamation and accurate accounting, we restricted the programming model for user-level shared objects somewhat: a shared heap's size is frozen after creation, and write barriers prevent the creation references to objects in user heaps. We have shown that despite these restrictions, applications can conveniently share complex data types such as hashmaps.

In our implementation, we modified an existing single-processor, user-mode Java virtual machine to support KaffeOS's functionality. Some implementation decisions, such as the use of class loaders for multiple namespaces or the specifics of how our garbage collector represents its per-heap data structures, were a result of our chosen infrastructure. However, KaffeOS's design principles apply to other JVMs as well, which is why we discussed them without assuming a concrete underlying JVM.

The core ideas of KaffeOS's design are the use of a red line for safe termination, the logical separation of memory in heaps for reclamation and resource control, the use of write barriers for resource control, the use of entry and exit items for separate garbage collection, and restricted direct sharing to support interprocess communication. These ideas do not have to be applied in their totality, but are useful individually as well. For instance, a red line could provide safe termination in JVMs that use a single heap; write barriers could support scoped memory as in Real-Time Java.

Finally, although KaffeOS was designed for a Java runtime system, its techniques should also be applicable to other languages that exploit type safety to run multiple, untrusted applications in a single runtime system. Such environments could also profit from KaffeOS's demonstrated ability to manage primary resources efficiently and the defenses against denial-of-service attacks it provides.

REFERENCES

BACK, G. 2002. Isolation, resource management, and sharing in the KaffeOS Java runtime system. Ph.D. dissertation. University of Utah School of Computing.

BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. 1998. Thin locks: Featherweight synchronization for Java. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Ont., Canada). ACM, New York, 258–268.

BALFANZ, D. AND GONG, L. 1998. Experience with secure multi-processing in Java. In *Proceedings of the 18th International Conference on Distributed Computing Systems* (Amsterdam, The Netherlands). 398–405.

BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Mass.). ACM, New York, 117–128.

BERGSTEN, H. 2000. *JavaServer Pages*, First ed. O'Reilly & Associates, Inc., Sebastopol, Calif.

BERNADAT, P., LAMBRIGHT, D., AND TRAVOSTINO, F. 1998. Towards a resource-safe Java for service guarantees in uncooperative environments. In *Proceedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications* (Madrid, Spain). IEEE Computer Society Press, Los Alamitos, Calif., 101–111.

BERSHAD, B., SAVAGE, S., PARDYAK, P., BECKER, D., FIUCZYNSKI, M., AND SIRER, E. 1995a. Protection is a software issue. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems* (Orcas Island, Wash.). 62–65.

BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. 1995b. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles* (Copper Mountain, Col.). 267–284.

BLACKBURN, S. M., SINGHAI, S., HERTZ, M., MCKINLEY, K. S., AND MOSS, J. E. B. 2001. Pretenuring for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)* (Tampa Bay, Fla.). ACM, New York, 342–352.

BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., AND TURNBULL, M. 2000. *The Real-Time Specification for Java*, First ed. The Java Series. Addison-Wesley, Reading, Mass.

CHAN, P., LEE, R., AND KRAMER, D. 1998. *The Java Class Libraries: Volume 1*, Second ed. The Java Series. Addison-Wesley, Reading, Mass.

CHASE, J. S. 1995. An operating system structure for wide-address architectures. Ph.D. dissertation. Department of Computer Science and Engineering, University of Washington, Seattle, Wash.

CHEN, H. AND WAGNER, D. 2002. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (Washington, D.C.). ACM, New York, 235–244.

CHENG, P., HARPER, R., AND LEE, P. 1998. Generational stack collection and profile-driven pretenuring. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Ont., Canada). ACM, New York, 162–173.

CZAJKOWSKI, G. 2000. Application isolation in the Java virtual machine. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '00)* (Minneapolis, Minn.). ACM, New York, 354–366.

CZAJKOWSKI, G., CHANG, C.-C., HAWBLITZEL, C., HU, D., AND VON EICKEN, T. 1998. Resource management for extensible Internet servers. In *Proceedings of the 8th ACM SIGOPS European Workshop* (Sintra, Portugal). ACM, New York, 33–39.

CZAJKOWSKI, G. AND DAYNÉS, L. 2001. Multitasking without compromise: A virtual machine evolution. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)* (Tampa Bay, Fla.). ACM, New York, 125–138.

Czajkowski, G. and von Eicken, T. 1998. JRes: A resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)* (Vancouver, B.C. Canada). ACM, New York, 21–35.

Dean, D. 1997. The security of static typing with dynamic linking. In *Proceedings of the 4th ACM Conference on Computer and Communications Security* (Zurich, Switzerland). ACM, New York, 18–27.

Dillenberger, D., Bordawekar, R., Clark, C. W., Durand, D., Emmes, D., Gohda, O., Howard, S., Oliver, M. F., Samuel, F., and John, R. W. S. 2000. Building a Java virtual machine for server applications: The JVM on OS/390. *IBM Syst. J. 39*, 1, 194–210.

Domani, T., Goldshtein, G., Kolodner, E. K., Lewis, E., Petrank, E., and Sheinwald, D. 2002. Thread-local heaps for Java. In *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany). ACM, New York, 183–194.

Dorward, S., Pike, R., Presotto, D. L., Ritchie, D. M., Trickey, H., and Winterbottom, P. 1997. The Inferno operating system. *Bell Labs Tech. J. 2*, 1, 5–18.

Engler, D., Chelf, B., Chou, A., and Hallem, S. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation* (San Diego, Calif.). USENIX Association, 1–16.

Engler, D. R., Kaashoek, M. F., and O'Toole Jr., J. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th Symposium on Operating Systems Principles* (Copper Mountain, Col.). 251–266.

Flatt, M. and Findler, R. 2004. Kill-safe synchronization abstractions. In *Proceedings of the SIGPLAN '04 Conference on Programming Language Design and Implementation* (Washington, D.C.). ACM, New York.

Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. 1996. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, Wash.). USENIX Association, 137–151.

Franz, M. 1997. Beyond Java: An infrastructure for high-performance mobile code on the World Wide Web. In *Proceedings of WebNet '97, World Conference of the WWW, Internet, and Intranet*, S. Lobodzinski and I. Tomek, Eds. Association for the Advancement of Computing in Education, Toronto, Ont., Canada, 33–38.

Gay, D. and Aiken, A. 1998. Memory management with explicit regions. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Ont., Canada). ACM, New York, 313–323.

Gorrie, L. 1998. Echidna—A free multiprocess system in Java. `http://www.javagroup.org/echidna/`.

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

Hawblitzel, C. and von Eicken, T. 2002. Luna: A flexible java protection system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, Mass.).

Henzinger, T., Jhala, R., Majumdar, R., Necula, G., Sutre, G., and Weimer, W. 2002. Temporal safety proofs for systems code. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)* (Copenhagen, Denmark).

Jaeger, T., Liedtke, J., and Islam, N. 1998. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium* (San Antonio, Tex.). USENIX Association, 143–157.

Java Apache Project. 2000. The Apache JServ project. `http://java.apache.org/jserv`.

Java Community Process. 2003. Jsr 121. `http://www.jcp.org/en/jsr/detail?id=121`.

Joy, B., Steele, G., Gosling, J., and Bracha, G. 2000. *The Java Language Specification*, Second ed. The Java Series. Addison-Wesley, Reading, Mass.

Jul, E., Levy, H., Hutchison, N., and Black, A. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 109–133.

Lepreau, J., Hibler, M., Ford, B., and Law, J. 1993. In-kernel servers on Mach 3.0: Implementation and performance. In *Proceedings of the 3rd USENIX Mach Symposium* (Santa Fe, N.M.). USENIX Association, 39–55.

LIANG, S. 1999. *The Java Native Interface: Programmer's Guide and Specification*, First ed. The Java Series. Addison-Wesley, Reading, Mass.

LIANG, S. AND BRACHA, G. 1998. Dynamic class loading in the Java virtual machine. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)* (Vancouver, B.C., Canada). ACM, New York, 36–44.

LIZT, J. 1999. Oracle JServer Scalability and Performance. `http://www.oracle.com/java/scalability/index.html?testresults_twp.html`. Java Products Team, Oracle Server Technologies.

MALKHI, D., REITER, M. K., AND RUBIN, A. D. 1998. Secure execution of Java applets using a remote playground. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy* (Oakland, Calif.). IEEE Computer Society Press, Los Alamitos, Calif., 40–51.

MARLOW, S., JONES, S. P., MORAN, A., AND REPPY, J. 2001. Asynchronous exceptions in haskell. In *Proceedings of the Conference on Programming Language Design and Implementation* (Snowbird, Ut.). ACM, New York, 274–285.

MCGRAW, G. AND FELTEN, E. 1997. *Java Security: Hostile Applets, Holes, and Antidotes*. Wiley Computer Publishing, New York.

MICROSOFT CORPORATION. 2003. .NET web pages. `http://msdn.microsoft.com/netframework/`.

PLAINFOSSÉ, D. AND SHAPIRO, M. 1995. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management (IWMM95)* (Kinross, Scotland). Lecture Notes in Computer Science, vol. 986. Springer-Verlag, New York, 211–249.

PRICE, D. W., RUDYS, A., AND WALLACH, D. S. 2003. Garbage collector memory accounting in language-based systems. In *Proceedings of 2003 IEEE Symposium on Security and Privacy* (Oakland, Calif.). IEEE Computer Society Press, Los Alamitos, Calif., 263–274.

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. 1980. Pilot: An operating system for a personal computer. *Commun. ACM 23*, 2, 81–92.

RITCHIE, D. M. AND THOMPSON, K. 1978. The UNIX time-sharing system. *The Bell Syst. Tech. J. 57*, 6 (July/Aug.), 1905–1930.

RIVEST, R. 1992. The MD5 message-digest algorithm. Internet Request for Comments RFC 1321, Internet Network Working Group. April.

ROSCOE, T. 1995. The structure of a multi-service operating system. Ph.D. dissertation. Queen's College, University of Cambridge, Cambridge, U.K.

RUDYS, A. AND WALLACH, D. S. 2002. Termination in language-based systems. *ACM Trans. Inf. Syst. Sec. 5*, 2 (May), 138–168.

SARASWAT, V. 1997. Java is not type-safe. `http://matrix.research.att.com/vj/bug.html`.

SAULPAUGH, T. AND MIRHO, C. A. 1999. *Inside the JavaOS Operating System*. The Java Series. Addison-Wesley, Reading, Mass.

SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (Seattle, Wash.). USENIX Association, 213–227.

SHAPIRO, J. S. 2003. Vulnerabilities in synchronous IPC designs. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (Oakland, Calif.). IEEE Computer Society Press, Los Alamitos, Calif., 251–262.

SIRER, E., FIUCZYNSKI, M., PARDYAK, P., AND BERSHAD, B. 1996. Safe dynamic linking in an extensible operating system. In *Proceedings of the 1st Workshop on Compiler Support for System Software* (Tucson, Az.). 141–148.

SPEC. 1998. SPEC JVM98 benchmarks. `http://www.spec.org/osg/jvm98/`.

STEENSGAARD, B. 2000. Thread-specific heaps for multi-threaded programs. In *Proceedings of the 2nd International Symposium on Memory Management* (Minneapolis, Minn.). ACM, New York, 18–24.

STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. 1999. Age-based garbage collection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)* (Denver, Col.). ACM, New York, 370–381.

SUGANUMA, T., OGASAWARA, T., TAKEUCHI, M., YASUE, T., KAWAHITO, M., ISHIZAKI, K., KOMATSU, H., AND NAKATANI, T. 2000. Overview of the IBM Java just-in-time compiler. *IBM Syst. J. 39*, 1, 175–193.

SWINEHART, D. C., ZELLWEGER, P. T., BEACH, R. J., AND HAGMANN, R. B. 1986. A structural view of the Cedar programming environment. *ACM Trans. Prog. Lang. Syst. 8*, 4 (Oct.), 419–490.

TULLMANN, P. A. 1999. The Alta operating system. M.S. dissertation, Department of Computer Science, University of Utah.

TULLMANN, P., HIBLER, M., AND LEPREAU, J. 2001. Janos: A Java-oriented OS for active network nodes. *IEEE J. Sel. Areas Commun. 19*, 3 (Mar.), 501–510.

TULLMANN, P. AND LEPREAU, J. 1998. Nested Java processes: OS structure for mobile code. In *Proceedings of the 8th ACM SIGOPS European Workshop* (Sintra, Portugal). ACM, New York, 111–117.

VAN DOORN, L. 2000. A secure Java virtual machine. In *Proceedings of the 9th USENIX Security Symposium* (Denver, Col.). USENIX Association, 19–34.

VAN DOORN, L., HOMBURG, P., AND TANENBAUM, A. S. 1995. Paramecium: an extensible object-based kernel. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems* (Orcas Island, Wash.). IEEE Computer Society Press, Los Alamitos, Calif., 86–89.

WAHBE, R., LUCCO, S., ANDERSON, T., AND GRAHAM, S. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles* (Asheville, N.C.). 203–216.

WALDSPURGER, C. A. 1995. Lottery and stride scheduling: Flexible proportional-share resource management. Ph.D. dissertation. Massachusetts Institute of Technology, Cambridge, Mass.

WICK, A., FLATT, M., AND HSIEH, W. 2002. Reachability-based memory accounting. In *Proceedings of the 2002 Scheme Workshop* (Pittsburgh, Pa.).

WILKINSON, T. 1996. Kaffe—A Java virtual machine. `http://www.kaffe.org/`.

WILKINSON, T., STIEMERLING, T., GULL, A., WHITCROFT, A., OSMON, P., SAULSBURY, A., AND KELLY, P. 1992. Angel: A proposed multiprocessor operating system kernel. In *Proceedings of the European Workshop on Parallel Computing* (Barcelona, Spain). 316–319.

WILSON, P. R. 1992. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management (IWMM92)*, Y. Bekkers and J. Cohen, Eds. Lecture Notes in Computer Science, vol. 637. Springer-Verlag, New York, 1–42.

WIND RIVER SYSTEMS, INC. 1995. *VxWorks Programmer's Guide*. Wind River Systems, Inc., Alameda, Calif.

WIRTH, N. AND GUTKNECHT, J. 1992. *Project Oberon*. ACM, New York.

ZEE, K. AND RINARD, M. 2002. Write barrier removal by static analysis. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '02)* (Seattle, Wash.). ACM, New York, 191–210.

Received April 2003; revised October 2003, April 2004; accepted May 2004