Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java

Godmar Back, Wilson C. Hsieh, Jay Lepreau
School of Computing
University of Utah

Abstract

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: inter-application memory protection and basic system services. They do not, however, provide the ability to isolate applications from each other, or limit their resource consumption. This paper describes KaffeOS, a Java runtime system that provides these features. The KaffeOS architecture takes many lessons from operating system design, such as the use of a user/kernel boundary, and employs garbage collection techniques, such as write barriers.

The KaffeOS architecture supports the OS abstraction of a *process* in a Java virtual machine. Each process executes as if it were run in its own virtual machine, including separate garbage collection of its own heap. The difficulty in designing KaffeOS lay in balancing the goals of isolation and resource management against the goal of allowing direct sharing of objects. Overall, KaffeOS is no more than 11% slower than the freely available JVM on which it is based, which is an acceptable penalty for the safety that it provides. Because of its implementation base, KaffeOS is substantially slower than commercial JVMs for trusted code, but it clearly outperforms those JVMs in the presence of denial-of-service attacks or misbehaving code.

1 Introduction

The need to support the safe execution of untrusted programs in runtime systems for type-safe languages has become clear. Language runtimes are being used in many environments for executing untrusted code: for example, applets, servlets, active packets [41], database queries [15], and kernel extensions [6]. Current systems (such as Java) provide *memory protection* through the enforcement of type safety and *secure system services* through a number of mechanisms, including namespace and access control. Unfortunately, malicious or buggy applications can deny service to other applications. For example, a Java applet can generate excessive amounts of garbage and cause a Web browser to spend all of its time collecting it.

To support the execution of untrusted code, type-safe language runtimes need to provide a mechanism to isolate and manage the resources of applications, analogous to that provided by operating systems. Although other resource management abstractions exist [4], the classic OS *process* abstraction is appropriate. A process is the basic unit of resource ownership and control; it provides isolation between applications. On a traditional operating system, untrusted code can be forked in its own process; CPU and memory limits can be placed on the process; and the process can be killed if it is uncooperative.

A number of approaches to isolating applications in Java have been developed by others over the last few years. An *applet context* [9] is an example of an application-specific approach. It provides a separate namespace and a separate set of execution permissions for untrusted applets. Applet contexts do not support resource management, and cannot defend against denial-of-service attacks. In addition, they are not general: applet contexts are specific to applets, and cannot be used easily in other environments.

Several general-purpose models for isolating applications in Java do exist, such as the J-Kernel [23] or Echidna [21]. However, these solutions superimpose an operating system kernel abstraction on Java without changing the underlying virtual machine. As a result, it is impossible in those systems to account for resources spent on behalf of a given application: for example, CPU time spent while garbage collecting a process's heap.

An alternative approach to separate different applications is to give each one its own virtual machine, and run each virtual machine in a different process on an un-

This research was largely 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. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

Contact information: {gback,wilson,lepreau}@cs.utah.edu. School of Computing, 50 S. Central Campus Drive, Room 3190, University of Utah, SLC, UT 84112-9205. http://www.cs.utah.edu/flux/

derlying OS [25, 29]. For instance, most operating systems can limit a process's heap size or CPU consumption. Such mechanisms could be used to directly limit an entire VM's resource consumption, but they depend on underlying operating system support.

Designing JVMs to support multiple processes is a superior approach. First, it reduces per-application overhead. For example, applications on KaffeOS can share classes in the same way that an OS allows applications to share libraries. Second, communication between processes can be more efficient in one VM, since objects can be shared directly. (One of the reasons for using type-safe language technology in systems such as SPIN [6] was to reduce the cost of IPC; we want to keep that goal.) Third, embedding a JVM in another application, such as a web server or web browser, is difficult (or impossible) if the JVM relies on an operating system to isolate different activities. Fourth, embedded or portable devices may not provide OS or hardware support for managing processes. Finally, a single JVM uses less energy than multiple JVM's on portable devices [19].

Our work consists of supporting processes in a modern type-safe language, Java. Our solution, KaffeOS, adds a process model to Java that allows a JVM to run multiple untrusted programs safely, and still supports the direct sharing of resources between programs. The difficulty in designing KaffeOS lay in balancing conflicting goals: process isolation and resource management versus direct sharing of objects between processes.

A KaffeOS process is a general-purpose mechanism that can easily be used in multiple application domains. For instance, KaffeOS could be used in a browser to support multiple applets, within a server to support multiple servlets, or even to provide a standalone "Java OS" on bare hardware. We have structured our abstractions and APIs so that they are as broadly applicable as possible, much as the OS process abstraction is. Because the KaffeOS architecture is designed to support processes, we have taken lessons from the design of traditional operating systems, such as the use of a user/kernel boundary.

Our design makes KaffeOS's isolation and resource control mechanisms comprehensive. We focus on the management of CPU time and memory, although we plan to address other resources such as network bandwidth. The runtime system is able to account for and control all of the CPU and memory resources consumed on behalf of any process. We have dealt with these issues by structuring the KaffeOS virtual machine so that it separates the resources used by different processes as much as possible.

To summarize, this paper makes the following contributions:

- We describe how lessons from building traditional operating systems can and should be used to structure runtime systems for type-safe languages.

- We describe how software mechanisms in the compiler and runtime can be used to implement isolation and resource management in a Java virtual machine.

- We describe the design and implementation of KaffeOS. KaffeOS implements our process model in Java, which isolates applications from each other, provides resource management mechanisms for them, and also lets them share resources directly.

- We show that the performance penalty for using KaffeOS is reasonable, compared to the freely available JVM on which it is based. Even though, due to that implementation base, KaffeOS is substantially slower than commercial JVMs on standard benchmarks, it outperforms those JVMs in the presence of uncooperative code.

Sections 2 and 3 describe and discuss the design and implementation of KaffeOS. Section 4 provides some performance measurements of KaffeOS, and compares its performance with that of some commercial Java virtual machines. Section 5 describes related work in more detail, and Section 6 summarizes our conclusions and results.

2 Design Principles

The following principles drove our design of KaffeOS, in decreasing order of importance:

- **Process separation**. We provide the "classical" property of a process: each process is given the illusion of having the whole virtual machine to itself.

- **Safe termination of processes**. Processes may terminate abruptly due to either an internal error or an external event. In both cases, we ensure that the integrity of other processes and the system itself is not violated.

- **Direct sharing between processes**. Processes can directly share objects in order to communicate with each other.

- **Precise memory and CPU accounting**. The memory and CPU time spent on almost all activities can be attributed to the application on whose behalf it was expended.

- **Full reclamation of memory**. When a process is terminated, its memory must be fully reclaimed. In a language-based system, memory cannot be revoked by unmapping pages: it must be garbage-collected. We restrict a process's heap writes to avoid uncollectable memory in the presence of direct object sharing.

- **Hierarchical memory management**. Memory allocation can be managed in a hierarchy, which provides a simple model for controlling processes.

The interaction between these design principles is complex. For expository purposes, we discuss these principles in a slightly different order in the remainder of this section.

Process separation. A process cannot accidentally or intentionally access another process' data, because each process has its own heap. A heap constitutes of a memory pool managed by an allocator and a garbage collector. Each process is given its own name space for its objects and classes, as well. Type safety provides memory protection, so that a process cannot access other process's objects.

To ensure process separation, an untrusted process is not allowed to hold onto system-level resources indefinitely. For instance, global kernel locks are not directly accessible to user processes. Violations of this restriction are instances of bad system design. Similarly, faults in one process must not impact progress in other processes.

Safe termination of processes. KaffeOS is structured such that critical parts of the system cannot be damaged when a process is terminated. For example, a process is not allowed to terminate when it is holding a lock on a system resource.

We divide KaffeOS into user and kernel parts [2], an important distinction used in operating system design. A user/kernel distinction is necessary to maintain system integrity in the presence of process termination.

Figure 1 illustrates the high-level structure of KaffeOS. User 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 (the rest of the runtime libraries and the garbage collector, as well as the virtual machine itself) must run in kernel mode to ensure their integrity. Note that "user mode" and "kernel mode" do not indicate a change in hardware privileges. Instead, they indicate different environments with respect to termination and resource consumption:

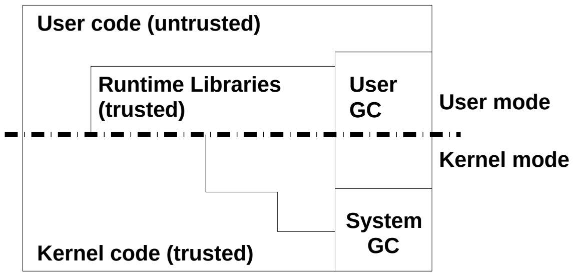

Figure 1: Structure of KaffeOS. System code is divided into kernel and user modes; user code all runs in user mode. In user mode, code can be terminated arbitrarily; in kernel mode, code cannot be terminated arbitrarily.

- Processes running in user mode can be terminated at will. Processes running in kernel mode cannot be terminated at an arbitrary time, because they must leave the kernel in a clean state.

- 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 entire system, although typically such use is charged to the appropriate user process.

Such a structure echoes that of exokernels [18], where system-level code executes as a user-mode library. Note that a language-based system allows the kernel to trust user-mode code to a great extent, because type safety prevents user code from damaging any user-mode system code.

The KaffeOS kernel is structured so that it can handle termination requests and internal errors cleanly. Termination requests are deferred, so that a process cannot be terminated while manipulating kernel data structures. Kernel code must not abruptly terminate due to internal exceptions, for the same reason. Violations of these two restrictions are considered kernel bugs.

Others have suggested that depending on language-level exception handling is sufficient for safe termination. We disagree, because exceptions interact poorly with code in critical sections, which leaves shared data structures open to corruption. Even if termination requests were deferred during critical sections, one would need transactional support to ensure the integrity of mutually related data structures in the absence of a kernel. In addition, such an approach would be a confusing overloading of the concepts of mutual exclusion and deferred termination; preventing termination while any lock is held would also violate isolation.

Full reclamation of memory. Since Java is type-safe, it does not provide a primitive to reclaim memory. Instead, unreachable memory is freed by a garbage collector. We use the garbage collector to recover all the memory of a process when it terminates. Therefore, we must prevent situations where the collector cannot free a terminated process's objects because another process still holds references to them.

We use techniques from distributed garbage collection schemes [31] to restrict cross-process references. Distributed GC mechanisms are normally used to overcome the physical separation of machines and create the impression of a global shared heap. We use distributed GC mechanisms to manage multiple heaps in a single address space, so that they can be collected independently.

We use *write barriers* [43] to restrict writes. A write barrier is a check that happens on every pointer write to the heap. As we show in Section 4, the cost of using write barriers, although non-negligible, is reasonable.

Illegal cross-references are those that would prevent a process's memory from being reclaimed: for example, references from one user heap to another. Since those references cannot exist, it is possible to reclaim a process's heap as soon as the process is terminated. Writes that would create illegal cross-references are forbidden, and raise exceptions. 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.

Unlike distributed garbage collection, in KaffeOS inter-heap cycles do not cause problems. The only form of inter-heap cycles that can occur are due to data structures that are split between a user heap and the kernel heap, since there can be no cycles that span multiple user heaps. Writes of user-heap references to kernel objects can only be done by trusted code. The kernel is coded so that it only writes a user-heap reference to a kernel object whose lifetime equals that of the user process: for example, the object that represents the process itself.

KaffeOS is intended to run on a wide range of systems. We assume that the platforms on which it runs will not necessarily have a hardware memory management unit under the control of KaffeOS. We also assume that the host may not have an operating system that supports virtual memory. For example, a Palm Pilot satisfies both of these assumptions. Under these assumptions, memory cannot simply be revoked by unmapping it.

Precise memory and CPU accounting. We account for memory and CPU on a per-process basis, so as to limit their consumption by buggy or possibly malicious code. In addition, to prevent denial-of-service attacks, it is necessary to minimize the amount of time and memory spent servicing kernel requests.

Memory accounting is complete. It applies not only to objects at the Java level, but to all allocations done in the VM on behalf of a given process. In contrast, bytecode-rewriting approaches that do not modify the virtual machine, such as Jres [13, 14], can only account for object allocations.

We try to minimize the number of objects that are allocated on the kernel heap through careful coding of the kernel interfaces. For instance, consider a system call that creates a new process with a new heap: the process object itself, which is large, is allocated on the new 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 only maintains a small entry in a process table.

We increase the accuracy of CPU accounting by minimizing the time spent in non-preemptible sections of code. In addition, separately collecting user heaps and the kernel heap reduces the amount of time spent in the kernel. We again use write barriers: here, to detect cross-references from a user to the kernel heap, and vice versa. For each such reference, we create an *entry item* in the heap to which it points [31]. In addition, we create a special *exit item* in the original heap to remember the entry item created in the destination heap. Unlike distributed object systems such as Emerald [26], entry and exit items are not used for naming non-local objects; we only use them to decouple the garbage collection of different heaps.

Entry items are reference counted: they keep track of the number of exit items that point to them. The reference count of an entry item is decremented when an exit item is garbage collected. If 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.

A process's memory is reclaimed upon termination by merging its heap with the kernel heap. All exit items are destroyed at this point and the corresponding entry items are updated. The kernel heap's collector can then collect all of the memory, including memory on the kernel heap that was kept alive by the process. User-kernel cycles of garbage objects can be collected at this time. Note that a user process could attempt to create and kill and large number of new heaps to deny service to other processes. Such an attempt can only prevented by imposing additional restrictions on the number or frequency with which a process may invoke kernel services.

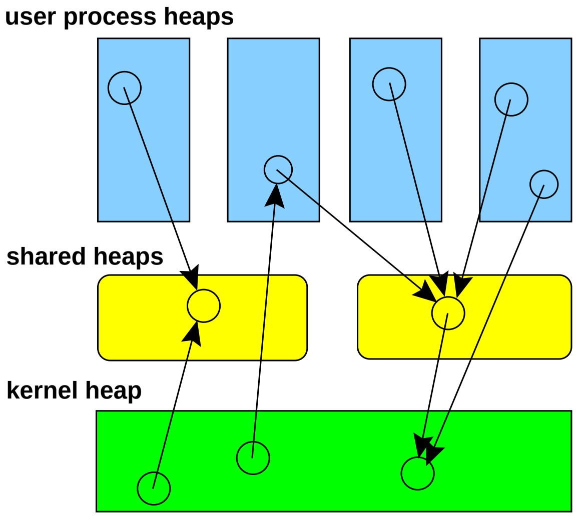

Figure 2: Heap structure in KaffeOS. The *kernel heap* can contain pointers into the user heaps, but the *shared heaps* and other user heaps cannot. User heaps can contain pointers into the kernel heap and shared heaps.

Direct sharing between processes. One of the reasons for using a language-based system is to allow for direct communication between applications. For example, the SPIN operating system allowed kernel extensions to communicate directly through pointers to memory. The design of KaffeOS retains this design principle. Figure 2 shows the different heaps in KaffeOS, and the kinds of inter-heap pointers that are legal.

In KaffeOS, a process can dynamically create a shared heap to communicate with other processes. A shared heap holds ordinary objects that can be accessed in the usual manner. Shared objects are not allowed to have pointers to objects on any user heap, because those pointers would prevent this user heap's full reclamation. This restriction is again enforced by write barriers; attempts to assign such pointers will result in an exception.

A shared heap has the following lifecycle. First, one process picks one or more shared types out of a central shared namespace, creates the heap, and loads the shared class or classes into it. While the heap is being created, the creator is charged for the whole heap. After the heap is populated with classes and objects, it is frozen and its size remains fixed for its lifetime. If other processes look up the shared heap, they are charged that amount. In this way, all sharers are charged for the heap. Processes exchange data by writing into and reading from the shared objects and by synchronizing on them in the usual way.

If a process drops all references to a shared heap, all exit items to that shared heap become 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 becomes orphaned. The kernel garbage collector checks for orphaned shared heaps at the beginning of each GC cycle and merges them into the kernel heap.

This model guarantees three properties:

- All sharers are charged in full for a shared heap while they are holding onto the shared heap, whose size is fixed. As a result, sharers do not have to be charged asynchronously if another sharer exits. (If n sharers were each to pay only $1/n$ of the cost of a shared heap, when one sharer exited the others would have to be asynchronously charged $(1/n - 1) - (1/n)$ of the cost.)

- As already discussed, one process cannot use a shared object to keep objects in another process alive.

- Sharers are charged accurately for all metadata, such as internal class data structures. The metadata is also allocated on the shared heap. Unfortunately, this prevents us from applying any optimization that allocates data structures related to the shared heap lazily during execution.

Although process heaps can be scanned independently during GC, thread stacks still need to be scanned during GC for inter-heap references. Incremental schemes could be used to eliminate repeated scans of a stack [12], and a thread does not need to be scanned more than once while it is suspended. Some "GC crosstalk" between processes is still possible, because a process could create many threads in an effort to get the system to scan them all. We decided that the benefit of allowing direct sharing between processes is worth leaving open such a possibility.

Hierarchical memory management. We provide a simple hierarchical model for managing memory. Each heap is associated with a *memlimit*, which consists of an upper limit and a current use. Memlimits form a hierarchy: each one has a parent, except for a root memlimit. All memory allocated to the heap is debited from that memlimit, and memory collected from that heap is credited to the memlimit. This process of crediting/debiting is applied recursively to the node's parents.

A memlimit can be *hard* or *soft*. 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. Credits and debits are therefore not propagated past a hard limit. A soft memlimit's maximum limit, on the other hand, is just a limit—credits and debits of a soft memlimit's current usage are reflected in the parent.

Hard and soft limits allow different memory management strategies. Hard limits allow for memory reservations, but incur inefficient memory use if the limits are not used. Memory consumption matters, because we do not assume there is an underlying operating system; as a result, KaffeOS may manage physical memory. Soft limits allow the setting of a summary limit for multiple activities without incurring the inefficiences of hard limits. They can be used to guard malicious or buggy applications where temporarily high memory usage can be tolerated.

Another application of soft limits is during the creation of shared heaps. Shared heaps are initially associated with a soft memlimit that is a child of the creating process heap's memlimit. In this way, they are separately accounted but still subject to their creator's memlimit, which ensures that they cannot grow to exceed their creator's ability to pay.

3 Discussion

The KaffeOS VM is built on top of the freely available Kaffe virtual machine, version 1.0b4 [42], which is roughly equivalent to JDK 1.1. In this section, we describe the specific issues that had to be dealt with in implementing KaffeOS. Many implementation decisions were driven by our desire to modify the Kaffe codebase as little as possible.

The primary purpose of KaffeOS is to run Java applications, which expect a well-defined environment of run-time services and libraries. We provide the standard Java API within KaffeOS.

We make use of various features of Java to support KaffeOS processes: Java class loaders, in particular, deserve some discussion. We also discuss our use of write barriers in more detail. Finally, we discuss some aspects of the Kaffe implementation that affect the performance that we can achieve with our KaffeOS prototype.

3.1 Write Barriers

An attempt to write a pointer to an object into a field of another object can have three different outcomes. In the common case, if a pointer to an object in the same heap is written, nothing needs to happen. If a pointer to a foreign heap is written, the write may either be aborted and trigger an exception, or it will cause the creation of a pair of exit/entry items to keep track of that allowable inter-heap reference.

The option of aborting writes ensures that the separation that is necessary for full reclamation is maintained. A write barrier exception is either related to a foreign user heap, or to a shared heap. If a pointer to a foreign user heap is written, such a pointer must have been passed on the stack or in a register as a return value from a kernel call. Such write barrier violations indicate kernel bugs, since the kernel is not supposed to return foreign references to a user process. Write barriers violations on the shared heap, on the other hand, indicate attempts by user code to create a connection from the shared heap to the user heap. Such attempts may either be malicious, or a sign of a violation of the programming model imposed on shared objects.

Keeping track of entry and exit items ensures the separation that is necessary for independent garbage collection and the garbage collection of shared heaps. In the third case, the write barrier code will maintain entry and exit items. As a result, a local garbage collector will know to include all incoming references as roots in its garbage collection cycle.

Independent garbage collection, which relies on accurate bookkeeping of entry and exit items, is important in our model. Therefore, write barriers are necessary, if only to maintain entry and exit items. This statement holds true even if no shared heaps are being used.

Write barriers could only be optimized away if their outcome is known. Such is the case within a procedure if static analysis reveals that an assignment is between pointers on the same heap (for instance, if newly constructed objects are involved), or that a previous assignment must have had the same outcome. In addition, if a generational collector were used, it should be possible to reduce the write barrier penalty by combining the code for the generational and the inter-heap write barrier checks.

3.2 Namespaces

Separate namespaces are provided in Java through the use of class loaders [28]. A class loader is an object that acts as a name server for types, and maps names to types. We use the Java class loading mechanism to provide KaffeOS processes with different namespaces. This use of Java class loaders is not novel, but is important because we have tried to make use of existing Java mechanisms when possible. When we use standard Java mechanisms, we can easily ensure that we do not violate the language's semantics.

Processes may share types for two reasons: either because the class in question is part of the run-time library (i.e., is a system or kernel class), or because it is the type of a shared object located on a shared heap, which must be identical in the processes that have access to the shared heap. We refer to the former as system-shared, and the latter as user-shared. Process loaders delegate the loading of all shared types to a shared loader. 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. Using one shared loader makes the namespace for user-shared classes global, which requires global and prior coordination between communicating partners. We use a simple naming convention for this shared namespace: the Java package `shared.*` contains all user-shared classes.

3.3 Java Class Libraries

To determine which classes can be system-shared, we examined each class in the Java standard libraries [10] to see how it interacted under the semantics of class loading. A class's members and their associated code are described by a sequence of bytes in a *class file*. Classes from identical class files that are loaded [28] by different class loaders are defined to be different in Java, even though they have identical behavior relative to the namespace defined by the loader that loaded them. We refer to such classes as *reloaded* classes. Reloaded classes are analogous to traditional shared libraries. Reloading a class gives each instance its own copies of static fields. In KaffeOS, Java classes could be reloaded; they could be modified to be shared across processes; or they could be used unchanged. For each class, we decided which alternative to choose, subject to two goals: to share as many classes as possible, but to make as few code changes as necessary.

Certain classes must be shared between 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! If a system-shared class uses static fields, and if these fields cannot be eliminated, they must be initialized with objects whose implementation is process-aware. Shared classes cannot directly refer to reloaded classes, because such references are represented using direct pointers by the run-time loader.

Non-shared classes should always be reloaded, so that each process gets its own instance. Reloaded classes do not share text in our current implementation, although they could. Because of some unfortunate decisions in the Java API design, some classes export static members as part of their public interface, which forces those classes to be reloaded. For example, `java.io.FileDescriptor` must be reloaded, because it exports the public static variables `in`, `out`, and `err` (stdin, stdout, and stderr, respectively). Other, possibly more efficient, ways to accomplish the same thing as reloading exist [16], but their impact on type safety is not fully understood. Out of roughly 600 classes in the core Java libraries, we are able to safely system-share about 430 (72%) of them. The rest of the classes are reloaded.

3.4 Java Language Issues

A few language compatability issues arose when building KaffeOS. For example, the Java language description assumes that all string literals are interned, and that equality can therefore be checked with a pointer comparison (the == operator). Unfortunately, to maintain such semantics, the interned string table would have to be a global (kernel) data structure—and user processes could allocate strings in an effort to make the kernel run out of memory. To deal with this problem, we chose to separately intern strings for each process. As a result, the Java language use of pointer comparison to check string equality does not work for strings that were created in different heaps, and the `equals` method must be used instead. It is impractical for the JVM to hide this semantic change from applications. However, this issue arises only in rare situations, and then only in KaffeOS-aware applications that directly use KaffeOS features.

3.5 Kaffe Limitations

Kaffe has relatively poor performance compared to commercial JVMs, for several reasons. First, its garbage collector is relatively primitive: it is a mark-and-sweep collector that is neither generational nor incremental. Second, it has a simple just-in-time bytecode compiler that translates each instruction individually. As a result, many unnecessary register spills and reloads are generated, and the native code that it produces is relatively poor.

4 Results

KaffeOS currently runs under Linux on the x86. We plan on porting it to the Itsy pocket computer from Compaq WRL; we have already ported Kaffe to the Itsy. To demonstrate the effectiveness of KaffeOS, we ran the following experiments:

- We measured three implementations of the write barrier. We ran the SPEC JVM98 benchmarks [35] on different configurations of KaffeOS, and the version of Kaffe on which it is based, and the IBM

JVM, which uses one of the fastest commercial JIT compilers [36] available. We must note that our results are not comparable with any published SPEC JVM98 metrics, as the measurements are not compliant with all of SPEC's run rules.

- We ran a servlet engine on KaffeOS to demonstrate that KaffeOS can prevent denial-of-service servlets from crashing a server. We also compared how the number of KaffeOS processes scales with how the number of OS processes scales.

Our measurements were all taken on a 800MHz "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 32K L1 cache, and combined 256K L2 cache.

4.1 Write Barrier Implementations

To measure the cost of write barriers in KaffeOS, we implemented several versions:

- *No Write Barrier*. We execute without a write barrier, and run everything on the kernel heap.
- *No Heap Pointer*. At each heap pointer write, the write barrier consists of a call to a routine that finds an object's heap ID by looking at the page on which the object lies and performs the barrier checks. In order to avoid cache conflict misses, the actual heap ID is stored in a block descriptor that is not on the same page. This implementation takes 37 cycles with a hot cache.
- *Heap Pointer*. At each heap pointer write, the write barrier consists of a call to a routine that finds an object's heap ID in the object header and performs the barrier checks. This implementation takes only 11 cycles with a hot cache, but adds 4 bytes per object.
- *Fake Heap Pointer*. To measure the impact of the 4 bytes of padding in the *Heap Pointer* implementation, we use the third barrier implementation but add 4 bytes to each object.

The KaffeOS JIT compiler does not yet inline the write barrier routine. Inlining the write barrier would not necessarily improve performance, as it would lead to substantial code expansion.

We ran the SPEC JVM98 benchmark suite on IBM's JVM, on Kaffe00 and on KaffeOS with different implementations of the write barrier. Kaffe00 is the code base upon which the current version of KaffeOS is built. This version is from June 2000. We instrumented Kaffe00 and KaffeOS to estimate how many cycles are spent during garbage collection. For IBM's JVM, we used a command line switch (-verbosegc) to obtain the number of milliseconds spent during garbage collection.

Figure 3 compares the results of our experiments. Each group of bars corresponds to IBM's JVM, Kaffe00, KaffeOS with no write barrier, KaffeOS with no heap pointer, KaffeOS with heap pointer, and KaffeOS with a fake heap pointer, in that order. The full bar displays the benchmark time as displayed by SPEC's JVM98 output. The upper part of the bar shows the time spent on those garbage collections that occurred during the actual benchmark run. Note that we excluded those collections that occurred while the SPEC test harness executed. The lower part of the bar represents the time not spent during garbage collection.

The time spent during garbage collection depends on the initial and maximum heap sizes, the allocation frequency, and the strategy used to decide when to collect. Kaffe00 and KaffeOS use a simple strategy: a collection is triggered whenever newly allocated memory exceeds 125% of the memory in use at the last GC. However, while Kaffe00 uses the memory occupied by objects as its measure, KaffeOS uses the number of pages as its measure, because KaffeOS's accounting mechanisms are designed to take internal fragmentation into account. In addition, KaffeOS decides when to collect for each heap separately. We do not know what strategy IBM's JVM uses, but its GC performance suggests that it is very aggressive at keeping its heap small.

Overall, IBM's JVM is between 2–5 times faster than Kaffe00; we will focus on the differences between Kaffe00 and the different versions of KaffeOS. While Kaffe00 and KaffeOS use different strategies for deciding when to collect, they use the same conservative non-moving collector. For this reason, we will focus on the time not spent on garbage collection.

The difference between Kaffe00 and KaffeOS *no write barrier* (excluding GC time) is minimal, which suggests that the changes done to Kaffe's run-time do not have significant performance impact. The difference between KaffeOS *no write barrier* and KaffeOS *no heap pointer* stems from the write barrier overhead, and is consistently below 7%.

Table 1 gives the number of write barriers that are executed in each of the SPEC benchmarks. When we compute the time to execute the write barriers by using the cycle counts for the barriers, we see that it is a fraction of the actual penalty. This discrepancy occurs because the microbenchmark uses a hot cache. For most benchmarks, the *heap pointer* optimization is effective in reducing the write barrier penalty to less than 5%. Excluding GC, KaffeOS *fake heap pointer* performs similarly

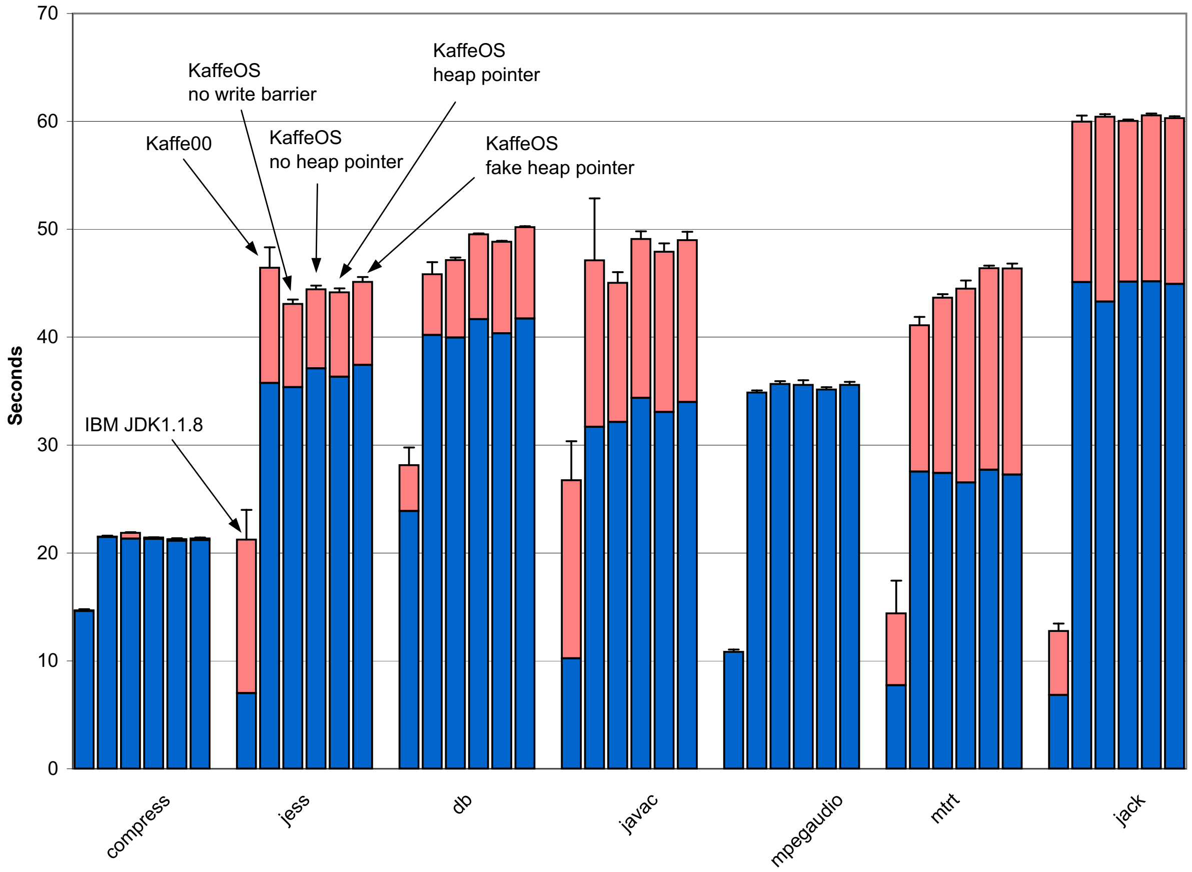

Figure 3: SPEC JVM98 run on various Java platforms. The error bars represent 95% confidence intervals. Each measurement is the result of three runs using SPEC's autorun mode. The upper part represents time spent in garbage collection.

Benchmark	Barriers	Time	Percent
compress	0.3M	0.014s	0.00%
jess	8.2M	0.38s	0.85%
db	30.4M	1.40s	2.84%
javac	21.1M	0.97s	1.97%
mpegaudio	5.8M	0.27s	0.75%
mtrt	3.3M	0.15s	0.34%
jack	20.2M	0.93s	1.54%

Table 1: Number of write barriers executed for each SPEC JVM98 benchmark. "Time" is the total CPU cycle cost for the write barrier instructions, assuming the *No Heap Pointer* cost of 37 cycles; "percent" is the fraction of the *No Write Barrier* execution time.

to KaffeOS *no heap pointer*; however, its overall performance is lower because more time is spent during GC.

On a better system with a more effective JIT, the relative cost of using write barriers would increase. On the other hand, a good JIT compiler could perform several kinds of optimizations to remove write barriers. A compiler should be able to remove redundant write barriers, along the lines of array bounds checking elimination. It could even perform method splitting to specialize methods, so as to remove useless barriers along frequently used call paths. We can only speculate as to what the performance penalty for implementing KaffeOS on the IBM JVM would be. Nevertheless, as we will show, the performance of KaffeOS is much better than that of the IBM JVM in the presence of uncooperative applications, despite the raw performance difference between them.

4.2 Servlet Engine

A Java servlet engine provides an environment for running Java programs (servlets) at a server. Their functionality subsumes that of CGI scripts at Web servers: for example, servlets may create dynamic content or run database queries. We use a **MemHog** servlet to measure the effects of a denial-of-service attack. MemHog sits in a loop, repeatedly allocates memory, and keeps it from being garbage-collected.

We compared KaffeOS's ability to prevent the MemHog servlet from denying service with that of IBM's JVM. We used Apache 1.3.12, JServ 1.1 (Apache's servlet engine), and a free version of JSDK 2.0 to run our tests, without modification. JServ runs servlets in *servlet zones*, which are virtual servers. A single JServ instance can host one or more servlet zones. We ran each JServ in its own KaffeOS process. We compared KaffeOS against IBM's JVM, in two configurations: one servlet zone per JVM (IBM/1), and multiple servlet zones in one JVM (IBM/n). Due to time constraints, we used an earlier version of KaffeOS for these benchmarks. This version is about half as fast as the version used for the SPEC JVM benchmarks.

When simulating this denial-of-service attack, we did what a system administrator concerned with availibility of his services would do: we restarted the JVM(s) and the KaffeOS process, respectively, whenever they crashed because of the effects caused by MemHog. In KaffeOS, MemHog will cause a single JServ to exit without affecting other JServs. If each JServ is started in its own IBM JVM, the whole JVM will eventually crash and be restarted. If all servlets are run in a single JServ on a single IBM JVM, the system runs out of memory in seemingly random places. This behavior resulted in exceptions that occurred at random places, which included the code that manipulated data structures that were shared between servlets in the surrounding JServ environment. Eventually, these data structures became corrupted, which results in an unhandled exception in JServ, or in some instances even a crash of the entire JVM.

Figure 4 illustrates the results of our experiments; note that the y axis uses a logarithmic scale. Running a separate KaffeOS process for each servlet has consistent performance, either with a MemHog running or without. This graph illustrates the most important feature of KaffeOS: that it can deliver consistent performance, even in the presence of uncooperative or malicious programs.

The graph shows that running each of the servlets in a single IBM JVM does not scale. This failure occurs because starting multiple JVMs eventually causes the machine to thrash. We estimate that each IBM JVM process takes about 2MB of virtual memory upon startup. We limited each JVM's heap size to 8MB in this configuration. An attempt to start 100 IBM JVMs rendered the machine inoperable.

If there are no uncooperative servlets running, using a single IBM JVM has the best performance. If there is a MemHog servlet running, such a configuration has worse performance than KaffeOS—despite the fact that KaffeOS is several times slower for individual servlets! This degradation is caused by a lack of isolation between servlets. However, as the ratio of well-behaved servlets to malicious servlets increases, the scheduler will yield less often to the malicious servlet. Consequently, the service of IBM/n,MemHog 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.

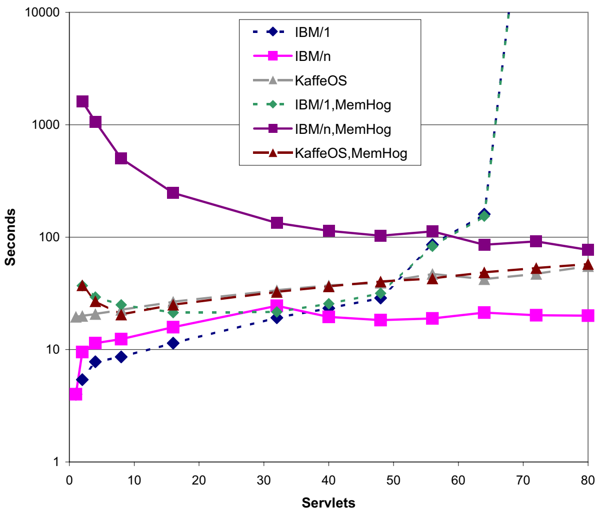

Figure 4: Scaling behavior of JVMs as the number of servlets increases. "IBM/1" means one IBM JVM per servlet; "IBM/n" means n servlets in one JVM. The "MemHog" measurements replace one of the good servlets with a MemHog. The y axis is the amount of time for the non-MemHog servlets to correctly respond to 1000 client requests.

Finally, we observe a slight service degradation as the number of KaffeOS processes increases. This degradation is likely due to inefficiencies in the user-mode threading system and scheduler.

5 Related Work

We classify the related work into three broad categories: extensible operating systems, resource management in operating systems, and Java extensions for resource management.

5.1 Extensible Operating Systems

Extensible operating systems have existed for many years. Most of them were not designed to protect against malicious users, although a number of them support strong security features. None of them, however, provides strong resource controls. Pilot [32] and Cedar [38] were two of the earliest language-based systems. Their development at Xerox PARC predates a flurry of research in the 1990's on such systems. These systems include Oberon [44] and Juice [20], which are based on the Oberon language; SPIN [6], which is based on Modula-3; and Inferno [17], which is based on a language called Dis. Such systems can be viewed as single-address-space operating systems (see Opal [11]) that use type safety for protection.

VINO is a software-based (but not language-based) extensible system [34] that addresses resource management by wrapping kernel extensions within transactions. When an extension exceeds its resource limits, it can be safely aborted (even if it holds kernel locks) and its resources can be recovered. Transactions are a very effective mechanism, but they are also relatively heavyweight.

5.2 Resource Management

Several operating systems projects have focused on quality-of-service issues and real-time performance guarantees. Nemesis [27] is a single-address-space OS that focuses on quality-of-service for multimedia applications. Eclipse [8] introduced the concept of a *reservation domain*, which is a pool of guaranteed resources. Eclipse provides a guarantee of cumulative service, which means that processes execute at a predictable rate. It manages CPU, disk, and physical memory. Our work is orthogonal, because we examine the software mechanisms that are necessary to manage computational resources.

Recent work on resource management has examined different forms of abstractions for computational resources. Banga et al. [4] describe an abstraction called *resource containers*, which are effectively accounts from which resource usage can be debited. Resource containers are orthogonal to a process' protection domain: a process can contain multiple resource containers, and processes can share resource containers. In KaffeOS we have concentrated on the mechanisms to simply allow resource management; resource-container-like mechanisms could be added in the future.

5.3 Java Extensions

Besides KaffeOS, a number of other research systems have explored (or are currently exploring) the problem of supporting processes in Java.

The J-Kernel [23] and JRes [13, 14] projects at Cornell explore resource control issues without making changes to the Java virtual machine. The J-Kernel extends Java by supporting capabilities between processes. These capabilities are indirection objects that can be used to isolate processes from each other. JRes extends the J-Kernel with a resource management interface whose implementation is portable across JVMs. The disadvantage of JRes (as compared to KaffeOS) is that Jres is a layer on top of a JVM; therefore, it cannot account for JVM resources consumed on the behalf of applications. Cornell is also exploring type systems that can support revocation directly [24].

Alta [39, 40] is a Java virtual machine that enforces resource controls based on a nested process model. The nested process model in Alta allows processes to control the resources and environment of other processes, including the class namespace. Additionally, Alta supports a more flexible sharing model that allows processes to directly share more than just objects of primitive types. Like KaffeOS, Alta is based on Kaffe, and, like KaffeOS, Alta provides support within the JVM for comprehensive memory accounting. However, Alta only provides a single, global garbage collector, so separation of garbage collection costs is not possible. For a more thorough discussion of Alta and the J-Kernel, see Back et al [1].

Balfanz and Gong [3] describe a multi-processing JVM developed to explore the security architecture ramifications of protecting applications from each other, as opposed to just protecting the system from applications. They identify several areas of the JDK that assume a single-application model, and propose extensions to the JDK to allow multiple applications and to provide inter-application security. The focus of their multi-processing JVM is to explore the applicability of the JDK security model to multi-processing, and they rely on the existing, limited JDK infrastructure for resource control.

Sun's original JavaOS [37] was a standalone OS written almost entirely in Java. It is described as a first-class OS for Java applications, but appears to provide a single JVM with little separation between applications. It was to be replaced by a new implementation termed "JavaOS for Business" that also ran only Java applications. "JavaOS for Consumers" is built on the Chorus microkernel OS [33] to achieve real-time properties needed in embedded systems. Both of these systems apparently require a separate JVM for each Java application, and all run in supervisor mode.

Joust [22], a JVM integrated into the Scout operating system [30], provides control over CPU time and network bandwidth. To do so, it uses Scout's path abstraction. However, Joust does not support memory limits on applications.

The Open Group's Conversant system [5] is another project that modifies a JVM to provide processes. It provides each process with a separate address range (within a single Mach task), a separate heap, and a separate garbage collection thread. Conversant does not support sharing between processes, unlike KaffeOS, Alta, and the J-Kernel.

The Real-Time for Java Experts Group [7] has published a proposal to add real-time extensions to Java. This proposal provides for *scoped memory* areas with a limited lifetime, which can be implemented using multiple heaps that resemble KaffeOS's heaps. The proposal also dictates the use of write barriers to prevent pointer assignments to objects in short-lived inner scopes. Real-Time Java's main focus is to ensure predictable garbage collection characteristics in order to meet real-time guarantees; it does not address untrusted applications.

6 Conclusions

We have described the design and implementation of KaffeOS, a Java virtual machine that supports the operating system abstraction of *process*. KaffeOS enables processes to be isolated from each other, to have their resources controlled, and still share objects directly. Processes enable the following important features:

- The resource demands of Java processes can be accounted for separately, including memory consumption and GC time.
- Java processes can be terminated if their resource demands are too high, without damaging the system.
- Termination reclaims the resources of the terminated Java process.

These features enable KaffeOS to run untrusted code safely, because it can prevent simple denial-of-service attacks that would disable standard JVMs. The cost of these features, relative to Kaffe, is reasonable. Because Kaffe's performance is poor compared to commercial JVMs, it is difficult to estimate the cost of adding such features to a commercial JVM—but we believe that the overhead should not be excessive. Finally, even though KaffeOS is substantially slower than commercial JVMs, it exhibits much better performance scaling in the presence of uncooperative code.

Acknowledgements

We thank many members of the Flux group, especially Patrick Tullmann, for enlightening discussion, comments on earlier drafts, and assistance in running some of the experiments. We would like to thank Tim Stack for his help in upgrading the KaffeOS code base to a more recent version of Kaffe, and Jason Baker for some earlier implementation work. We are also grateful to those anonymous referees whose questions and comments helped us improve the paper, and to our shepherd David Culler.

References

[1] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *Proc. of the USENIX 2000 Annual Technical Conf.*, pages 197–210, San Diego, CA, June 2000. USENIX Association.

[2] G. V. Back and W. C. Hsieh. Drawing the red line in Java. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, Mar. 1999. IEEE Computer Society.

[3] D. Balfanz and L. Gong. Experience with Secure Multi-Processing in Java. In *Proc. of the Eighteenth International Conf. on Distributed Computing Systems*, May 1998.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, Feb. 1999.

[5] P. Bernadat, L. Feeney, D. Lambright, and F. Travostino. Java Sandboxes meet Service Guarantees: Secure Partitioning of CPU and Memory. Technical Report TOGRI-TR9805, The Open Group Research Institute, June 1998.

[6] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, CO, Dec. 3–6, 1995.

[7] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[8] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *Proc. of the 1998 USENIX Annual Technical Conf.*, pages 235–246, New Orleans, LA, June 1998.

[9] P. Chan and R. Lee. *The Java Class Libraries: Volume 2*. The Java Series. Addison-Wesley, second edition, November 1998.

[10] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries: Volume 1*. The Java Series. Addison-Wesley, second edition, November 1998.

[11] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, 1994.

[12] P. Cheng, R. Harper, and P. Lee. Generational Stack Collection and Profile-Driven Pretenuring. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 162–173, Montreal, Canada, June 1998.

[13] G. Czajkowski, C.-C. Chang, C. Hawblitzel, D. Hu, and T. von Eicken. Resource Management for Extensible Internet Servers. In *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, Sept. 1998.

[14] G. Czajkowski and T. von Eicken. JRes: A Resource Accounting Interface for Java. In *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, Oct. 1998. ACM.

[15] Delivering the Promise of Internet Computing: Integrating Java With Oracle8*i*. http://www.oracle.com/database/documents/-delivering_the_promise_twp.pdf, Apr. 1999.

[16] D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. S. John. Building a Java virtual machine for server applications: The JVM on OS/390. *IBM Systems Journal*, 39(1):194–210, 2000. Reprint Order No. G321-5723.

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

[18] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Dec. 1995.

[19] K. Farkas, J. Flinn, G. Back, D. Grunwald, and J. Anderson. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. In *Proceedings of SIGMETRICS '00*, page to appear, Santa Clara, CA, June 2000.

[20] M. Franz. Beyond Java: An Infrastructure for High-Performance Mobile Code on the World Wide Web. In S. Lobodzinski and I. Tomek, editors, *Proceedings of WebNet '97*, pages 33–38, October 1997.

[21] L. Gorrie. Echidna — A free multiprocess system in Java. http://www.javagroup.org/echidna/.

[22] J. H. Hartman et al. Joust: A Platform for Communication-Oriented Liquid Software. Technical Report 97–16, Univ. of Arizona, CS Dept., Dec. 1997.

[23] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. In *Proc. of the USENIX 1998 Annual Technical Conf.*, pages 259–270, New Orleans, LA, June 1998.

[24] C. Hawblitzel and T. von Eicken. Type System Support for Dynamic Revocation. In *Second Workshop on Compiler Support for System Software*, Atlanta, GA, May 1999.

[25] T. Jaeger, J. Liedtke, and N. Islam. Operating System Protection for Fine-Grained Programs. In *Proc. of the Seventh USENIX Security Symposium*, pages 143–157, Jan. 1998.

[26] E. Jul, H. Levy, N. Hutchison, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[27] I. M. Leslie, D. McAuley, R. J. Black, T. Roscoe, P. R. Barham, D. M. Evers, R. Fairbairns, and E. A. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.

[28] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications '98*, pages 36–44, Vancouver, Canada, Oct. 1998.

[29] D. Malkhi, M. K. Reiter, and A. D. Rubin. Secure Execution of Java Applets using a Remote Playground. In *Proc. of the 1998 IEEE Symposium on Security and Privacy*, pages 40–51, Oakland, CA, May 1998.

[30] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 153–167, Seattle, WA, Oct. 1996. USENIX Association.

[31] D. Plainfossé and M. Shapiro. A Survey of Distributed Garbage Collection Techniques. In *Proceedings of the 1995 International Workshop on Memory Management*, Kinross, Scotland, Sept. 1995.

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

[33] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus Distributed Operating System. *Computing Systems*, 1(4):287–338, Dec. 1989.

[34] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, WA, Oct. 1996. USENIX Association.

[35] SPEC JVM98 Benchmarks. http://www.spec.org/-osg/jvm98/.

[36] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[37] Sun Microsystems, Inc. JavaOS: A Standalone Java Environment, Feb. 1997. http:-//www.javasoft.com/products/javaos/javaos.-white.html.

[38] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, Oct. 1986.

[39] P. Tullmann and J. Lepreau. Nested Java Processes: OS Structure for Mobile Code. In *Proc. of the Eighth ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, Sept. 1998.

[40] P. A. Tullmann. The Alta Operating System. Master's thesis, University of Utah, 1999. 104 pages. Also available at http://www.cs.utah.edu/-flux/papers/tullmann-thesis-abs.html.

[41] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of IEEE OPENARCH '98*, San Francisco, CA, April 1998.

[42] T. Wilkinson. Kaffe – a Java virtual machine. http://www.transvirtual.com.

[43] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the 1992 International Workshop on Memory Management*, St. Malo, France, Sept. 1992. Expanded version, submitted to Computing Surveys.

[44] N. Wirth and J. Gutknecht. *Project Oberon*. ACM Press, New York, NY, 1992.