Garbage Collector Memory Accounting in Language-Based Systems

David W. Price Algis Rudys Dan S. Wallach
Department of Computer Science, Rice University
{dwp,arudys,dwallach}@cs.rice.edu

Abstract

Language run-time systems are often called upon to safely execute mutually distrustful tasks within the same runtime, protecting them from other tasks' bugs or otherwise hostile behavior. Well-studied access controls exist in systems such as Java to prevent unauthorized reading or writing of data, but techniques to measure and control resource usage are less prevalent. In particular, most language run-time systems include no facility to account for and regulate heap memory usage on a per-task basis. This oversight can be exploited by a misbehaving task, which might allocate and hold live enough memory to cause a denial-of-service attack, crashing or slowing down other tasks. In addition, tasks can legitimately share references to the same objects, and traditional approaches that charge memory to its allocator fail to properly account for this sharing. We present a method for modifying the garbage collector, already present in most modern language run-time systems, to measure the amount of live memory reachable from each task as it performs its regular duties. Our system naturally distinguishes memory shared across tasks from memory reachable from only a single task without requiring incompatible changes to the semantics of the programming language. Our prototype implementation imposes negligible performance overheads in a variety of benchmarks, yet provides enough information for the expression of rich policies to express the limits on a task's memory usage.

1 Introduction

Multitasking language run-time systems appear in a variety of commercial systems, ranging from applets running in web browsers and servlets in web servers to plugins running in extensible databases and agents running in online markets. By enforcing the language's type system and restricting the interfaces available to untrusted code, language-based systems can often achieve very restrictive "sandbox" policies, limiting access to the file system, network, and other specific resources. Furthermore, by avoiding the costs of separate operating system processes, the costs of context switching overhead and inter-task communication can be radically reduced. Since all tasks share the same address space, pointers to objects can be directly passed from one task to another. Tasks should be thought of as a generalization of Java applets or servlets; a task is both the code and the data on which the code operates.

1.1 Availability

When running multiple concurrent tasks which might be actively malicious to each other in a single language runtime system, it is critical for the system to make guarantees about its availability. Access controls are not sufficient to protect against denial of service attacks. More generally, most existing security mechanisms built into language-based systems focus on *safety* policies, that is, security policies that are strictly a function of a task's prior execution history and the current request being made. Such policies are designed to guarantee that "nothing bad ever happens." However, to preserve system availability, we need policies that talk about the future, i.e., "something good eventually happens." Such *liveness* policies [2] can cover topics as diverse as preventing deadlocks or preventing exhaustion of aggregate resources including CPU, memory, and network usage. In these cases, we cannot conclusively state, for any given lock operation or memory allocation, whether the program is in a "safe" state. However, we do wish to guarantee that the system won't get stuck.

When designing systems to have high availability, there are two general approaches we can take. One approach is to dynamically monitor system usage to detect when bad conditions occur, and taking corrective action when necessary. The other approach is to statically analyze the system and prove that the it cannot ever reach a bad state (or, that it will always, eventually, reach a good state). For this research, we are concerned with the availability of free memory, for which researchers have designed both static and dynamic mechanisms (related work and comparable systems are discussed in Sections 2 and 5). Since we wish

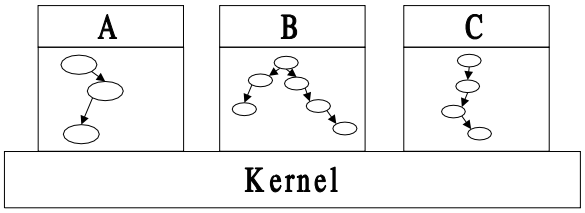

Figure 1. In traditional operating systems, each process is its own, self-contained box, so determining the memory usage of a process is equivalent to measuring the size of the box.

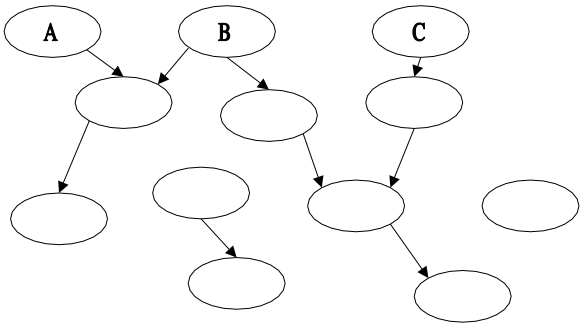

Figure 2. In language-based systems, memory may be shared among multiple tasks, so determining the memory usage of a single task is difficult.

to support general-purpose programming languages, computing static bounds on memory usage would be computationally infeasible. Instead, we need a mechanism that can efficiently measure memory usage, even in the presence of data sharing across tasks and with garbage collection managing the memory. Given such a measurement mechanism, we can then envision policies that detect when the system is running out of free memory. Using the measured data, these policies could identify misbehaving tasks and terminate them, reclaiming their resources, while allowing other tasks to continue running uninterrupted.

1.2 Language-based systems

In a traditional operating system, the problem of measuring memory allocation is straightforward. Processes encapsulate all the memory being used by a given task, making it easy to measure the total memory in use and to apply limits on how big a process can grow (see Figure 1). Likewise, when a process is terminated, it is easy to reclaim all the memory in use because it is part of the process's address space. To achieve this containment of memory, process-structured systems limit the ability to share data, typically requiring objects to be copied rather than shared by reference. Ideally, we would like to have the low-cost, type-safe sharing that can be achieved in language-based systems (e.g., see Figure 2) combined with the memory accounting and termination semantics achievable with process-structured systems. Currently, however, language-based systems lack semantics for measuring their tasks' memory usage. We wish to support arbitrary sharing of object references across tasks, implying that more than one task may be sharing the responsibility of keeping any given object live. This complicates any definition of memory usage, as some objects could potentially be "counted" more than once.

Most language-based systems use a garbage collector to provide memory management services. By design, the garbage collector already examines the entire heap to discover what memory is being held live. By making simple modifications to the garbage collector, causing it to process each task in turn and count as it goes, we can track the memory usage of individual tasks with little overhead beyond the regular cost of garbage collection.

Our system provides sufficient information to allow for a variety of flexible memory usage policies. The system maintains statistics not only on the amount of memory a task uses but also on the degree to which that memory is shared with other tasks. This would enable such policies as limiting the amount of memory used exclusively by a task to some percentage of system memory, and likewise limiting the total amount of memory used by the task. Language-based mechanisms like soft termination [46] could then be used to enforce such policies, terminating any tasks that violate the limit.

In the following sections, we describe the design and implementation of our memory accounting system. We describe the design of our system in Section 2. Section 3 discusses our implementation of memory accounting and its performance impact. We discuss the sorts of policy semantics our system supports in Section 4. Finally, we present related work in Section 5, and future work in Section 6.

2 Design

There are several different ways for a language-based system to track the memory usage of its individual tasks. We first discuss some proposed solutions, and describe the hard problems raised by their failings. We then discuss the design of our system.

2.1 Instrumented allocation

One common mechanism for determining the memory usage of tasks is to rewrite the task's code at load time. Memory allocations are instrumented to charge tasks with memory usage when they allocate objects, granting rebates when those objects are finalized. This approach has the benefit that no modifications are required to the underlying language run-time system. JRes [22] and Beg and Dahlin [8] both instrument memory allocations as a way to account for memory usage by tasks in Java.

However, there are several problems to using this approach. First, only allocation that explicitly occurs in the task is charged to that task. Any implicit allocation or allocation performed by the system on behalf of the task is not charged to it. In addition, in both JRes and Beg and Dahlin's system, accounting is performed on a per-thread basis. If a "system" thread or a another task's thread calls into a task, it could potentially be "tricked" into allocating memory on behalf of the task, giving that memory away "for free."

Furthermore, tasks can share memory with one another (see Figure 2). A task may allocate a block of memory, share that memory with another task, and later drop its pointer. In most language-based systems, however, memory is kept alive if any live pointers to it exist. As a result, another task could, out of necessity or malice, hold memory live; the task that initially allocated that memory would be forced to keep paying for it.

2.2 Process abstractions

Another common mechanism for accounting for memory usage is to use process abstractions. In some systems, each task is allocated its own heap, and the memory usage charged to that task is the size of that heap. KaffeOS [5, 6] is a system for Java that, in conjunction with an explicit process-like abstraction for Java tasks, provides a separate heap for each task. The multitasking virtual machine (MVM) [21] and systems by Bernadat et al. [9], and van Doorn [51] similarly use separate heaps or memory spaces to facilitate accounting for memory. Some systems [40, 47] even go so far as to run the JVMs in separate Unix processes on separate machines.

These systems accurately account for memory a task keeps live. However, inter-task communications and memory sharing are severely restricted, limiting the usefulness of the language. In addition, these systems are implemented with nontrivial customizations to the VM. Adapting these ideas to a new VM can require significant engineering resources.

In some systems, function calls and memory references are artificially restricted (either through some mechanism built into the run-time system or using code-to-code transformations). In this case, instrumenting memory allocations and object finalization yields an accurate accounting for the amount of memory used by a task. Examples include J-Kernel [34], J-SEAL2 [11], and Luna [35]. These systems are more accurate than strictly instrumented allocation. However, they still restrict inter-task communications and memory sharing among tasks.

2.3 Garbage collection-based accounting

Once we allow object references to be shared across tasks, the task that allocates an object in memory may not necessarily be the task that ends up using the object or keeping it live. Once a reference to an object has been given out, anybody could potentially hold that object reference. Clearly, we would like to only charge tasks for the memory they are keeping live, rather than the memory they allocate.

Under this rationale, live objects should be charged to those tasks from which they are reachable in the graph of heap objects. Conveniently, tracing garbage collectors already traverse this graph to find the reachable objects and free the space occupied by unreachable objects. By carefully managing the order in which the GC does its work and having the GC report back to us on its findings, we can use the GC as our tool for measuring each task's live memory footprint.

A typical garbage collector works by starting at a defined root set of references and doing a graph traversal to find all the memory reachable from those references. Memory not reached during this graph traversal is garbage and can be used for allocating new objects. In our system, we augment the collector to sequentially trace all the reachable memory from each task's root set.

The root set of a task is a set of roots in memory defined to be affiliated with that task; for example, our implementation defines it to be the static fields of all the task's classes plus the execution stacks of all its threads. For each task, the collector traces all reachable memory from its root set. As it does so, it computes the sum of the sizes of the objects it has seen. Once the traversal is complete, that sum is charged to the task currently being processed. Once the collector finishes iterating over all the tasks, it makes one final pass, starting with the set of all roots not affiliated with any particular task. Any objects which have not yet been reached after this completes are unreachable.

2.3.1 Handling shared memory

Because each object is only processed once in each garbage collection cycle, this method will find less and less shared memory as it goes from the start to the end of the list of tasks. As indicated by Figure 3, the collector will find all the memory that the first processed task shares with others, and

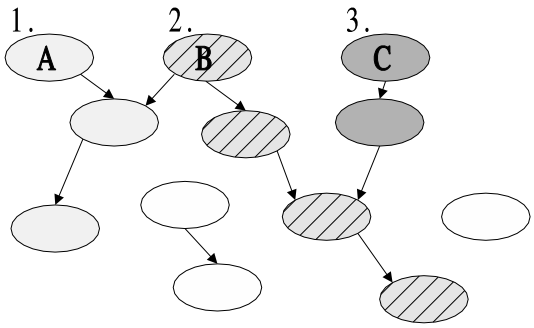

Figure 3. On a pass through the garbage collector, the first task to be scanned (in this case, task *A*) is charged for all memory reachable from it, while the last task scanned (in this case, task *C*) is charged for memory reachable only from it.

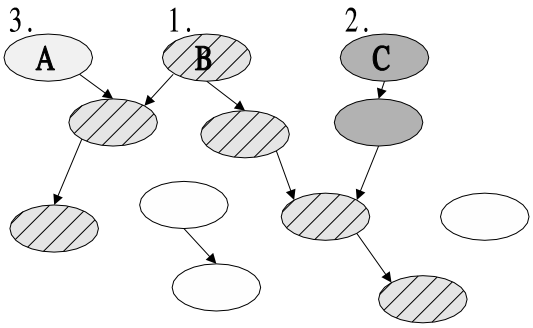

Figure 4. On subsequent invocations of the scan, the order of scanning is rotated; task *A*, the first task scanned in the previous example, is the last task scanned in this example. This process gives a range of memory usages for a task, including a maximum (all memory reachable from the task) and a minimum (memory reachable only from the task).

none of the memory shared by the last one processed. This asymmetry presents a problem: since the scanning mechanism treats each task's shared memory differently, we get an inconsistent view of the memory usage picture.

One option would be to run the garbage collector separately for each task. This would return the total amount of memory being held live by each task, similar to the "precise" policy described by Wick et al. [53]. Since we would need to start garbage collection fresh for each task, such a system would impose additional time costs, processing each shared object once for every task that can reach it. Without additional computation (and additional overhead), this system would also not report how much of the memory used by a given task is shared with other tasks.

We instead chose to address the asymmetry problem by rotating the order that tasks are processed on subsequent collections. The effect of this can be seen by comparing Figures 3 and 4; changing the processing order changes the memory charged to each task.

The first task processed yields a maximum value—an upper bound on memory reachable by that task and includes all memory it shares with other tasks. The last task processed yields a minimum value, indicating how much memory that task is responsible for that no other task has a reference to. Results for tasks in the middle give an intermediate value somewhere between these two extremes. Rotating tasks from the back of the processing list to the front means that the minimum and maximum values computed for each will be measured one collection apart from each other. This yields an imperfect snapshot of memory usage, but barring dramatic swings in memory being held live by a task in between collections, this rotation gives a valuable approximation to how much memory each task is both using on its own and is sharing with other tasks. The synthesis of this raw information into useful policies is discussed further in Section 4.

2.3.2 Unaccountable references

One concern of garbage collector-based memory accounting has been described as the "resource Trojan horse" problem [35]. In this case, task *B* might accept a reference to an object provided by task *A*. This object might in turn contain a reference to a very large block of memory. Task *B* will then be held responsible for that large block, even if it is unaware of the block's existence. Depending on the system's memory management policy, this could represent a denial of service attack on task *B*. Task *B* may want to accept a reference to an object controlled by an untrusted task without exposing itself to such an attack. Similarly, a system library providing access to a database may (generously) not want the client task to be charged for storage within the database. Finally, it might be the case that all tasks have pointers to and from a centralized manager system, and so there is a path in memory from each task to the memory of every other task. Our system as described so far would naïvely follow these references and describe the whole system as one region being fully shared among all the tasks. This is clearly not the most insightful view of the picture; we want a way to support all these styles of references, yet still be able to separate tasks from one another for measurement of their memory usage.

We solve these issues by introducing *unaccountable references*. Analogous to a weak reference (which refers to some data without holding it live), this type of reference refers to data, holds it live, but prevents the referrer from having to pay for what's on the other side. In our system, when the garbage collector encounters an unaccountable reference, it stops without proceeding to the object being referred to. After all tasks are processed, it starts again with all the unaccountable references in the system as roots. If the only path a task has to some memory is through an unaccountable reference, the memory will be guaranteed to be held live, but that task will not be charged for that memory.

A task must not be able to use unaccountable references to circumvent the memory accounting system. One solution would be to use language-level access control (e.g., stack inspection) to restrict the creation of unaccountable references to privileged code.

Our preferred approach is to permit any task to create an unaccountable reference, but to tag that reference with its creator's name. When these references are processed, the accounting system charges the memory found to the reference's creator. This technique, implemented as a small adjustment to the accounting system, nonetheless provides powerful semantics for memory sharing; a task can provide references to some service it's providing, and make it explicit in the interface that clients will not be billed for the memory found on the other side of the reference.

2.3.3 Generational GC

Generational garbage collection presents some challenges to our system. In a generational system, not all objects are traced every time the GC system is invoked. Instead, objects are allocated into a "nursery" heap, and are tenured by frequent minor collections into a mature space, which is collected using some other algorithm when it fills. Memory in the nursery is transient: upon each collection, it is either tenured or reclaimed. Thus, we're primarily interested in accounting for the memory that makes it to the mature space.

We can track a task's mature heap memory usage in two ways. When the mature space fills, we do a major collection and count mature heap memory that remains alive using the techniques described above in Section 2.3.1. When the nursery fills, a minor collection is performed. As objects are tenured, the size of each object is added to the total memory used by the task. At each major collection, this additive component is reset to zero. Thus, every tenured object will have been charged to one of the tasks that held it live while it was in the nursery. On subsequent major collections, the tasks holding the object live will share the cost of the object in the same fashion as they do for non-generational semispace collectors.

2.3.4 Other memory management techniques

We have implemented our system in a standard semispace collector and a generational collector, but we anticipate that it can be made to work with most precise, tracing collectors. In particular, we expect that it would map well to mark-and-sweep collectors, as this class of garbage collector also traces through memory finding live objects from a defined set of roots.

Our approach would not work if the memory management system used reference counting. Such a system does not do graph traversals over the space of objects in the heap, and so it would not discover the pattern of objects being held live by various roots, nor could it make any meaningful inferences about memory sharing.

A conservative garbage collector [12] would raise a number of difficult issues: tasks might be charged for memory discovered when the collector follows something that is not actually a reference. Unaccountable references would likely cause a significant performance hit, as each reference followed, unable to explicitly describe itself as an unaccountable reference, would have to be checked against a table of such references.

3 Implementation and results

We implemented our design in Java using IBM's Jikes Research Virtual Machine (RVM) [1] version 2.1.0. We found the RVM to be extremely useful for our work: it is implemented in Java, is largely self-hosting (e.g., the garbage collectors are, themselves, written in Java), and provides several different garbage collectors to choose from. We implemented our system as a set of changes to the RVM's simple copying collector (called "semispace") and its two-generational collector ("copyGen"). GCTk[1] is a flexible garbage collection toolkit for the RVM, but we chose to work with the default GC system that ships with the RVM as it satisfied our requirements.

The set of changes that we made to the RVM codebase is small; our changes can be expressed as a thousand-line patch against the original 64,000-line RVM codebase. Our modified RVM exposes additional functionality to allow the system to label which classes and threads are associated with which tasks, and to query the resource usage statistics of any given task. The resulting RVM is fully backwards-compatible with the original.

For the purposes of our prototype implementation, we defined a task to be a set of classes loaded by a particular `ClassLoader` instance, plus any threads that loaded those classes, plus those threads' children. The root set of each task processed by the garbage collector consists of the

[1] See `http://www.cs.umass.edu/~gctk/`.

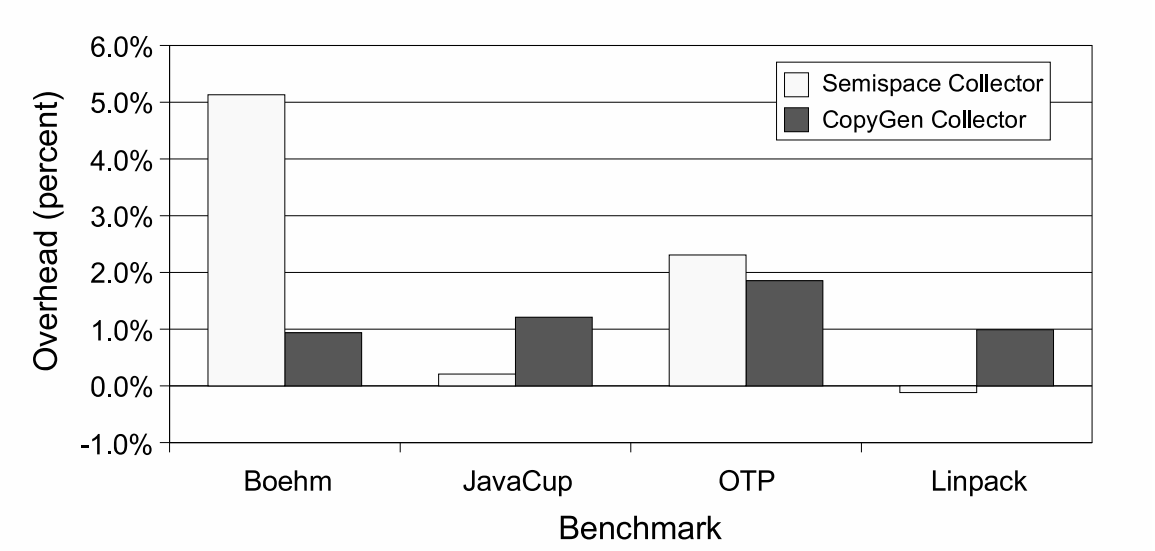

Figure 5. Runtime overhead incurred by the accounting modifications on Boehm's artificial GC benchmark and on various real-world application benchmarks with the RVM "semispace" and "copyGen" collectors.

static fields of all of its classes and the stacks of all of its threads.

We benchmarked our implementation on a 1 GHz AMD Athlon with 512MB of memory running version 2.4.18 of the Linux kernel. Different benchmarks allocate different amounts of memory, so we chose heap size appropriately in order to guarantee that the tasks would execute without allocation errors but still exercise the garbage collector sufficiently as to measure our modified system's performance.

3.1 Boehm microbenchmark

We wanted to ensure that our modifications to the garbage collector did not adversely impact its performance, so we benchmarked our implementation using Hans Boehm's artificial garbage collection benchmark[2], which repeatedly builds up and throws away binary trees of various sizes. Figure 5 shows the overhead of memory accounting for the two garbage collectors we used on this benchmark. The results indicate that the modified GC incurs a small percentage of overhead as a cost of doing its accounting.

3.2 Application benchmarks

We also benchmarked some real-world Java applications to get a sense of the overhead for programs not specifically designed to stress-test the memory subsystem. One limitation we suffered was that AWT is not yet implemented with the RVM, so we were somewhat limited in our choice of programs. We benchmarked the applications JavaCup,[3] a LALR parser-generator, Linpack,[4] an implementation of a matrix multiply, and OTP,[5] an S/Key-style one-time-password generator.

Figure 5 shows the overhead of our memory accounting system for these three applications for the two garbage collection systems. As with the Boehm microbenchmark, the slowdown is negligible. In the case of Linpack with the semispace collector, we actually saw a minuscule speedup. Linpack puts very little pressure on the GC system, allocating large arrays once, then processing with no further allocation. It's thus unsurprising that our changes to the GC system have minimal impact. A small speedup could result from fortuitous rearrangements of how code or data collides in the processor's caches, TLB, and so forth.

3.3 Multitasking microbenchmark

Since our system is designed to handle several tasks running concurrently, sharing memory amongst themselves, traditional single-tasking benchmarks are insufficient to ex-

[2] `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/`.

[3] `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[4] `http://netlib2.cs.utk.edu/benchmark/linpackjava/`.

[5] `http://www.cs.umd.edu/~harry/jotp/`.

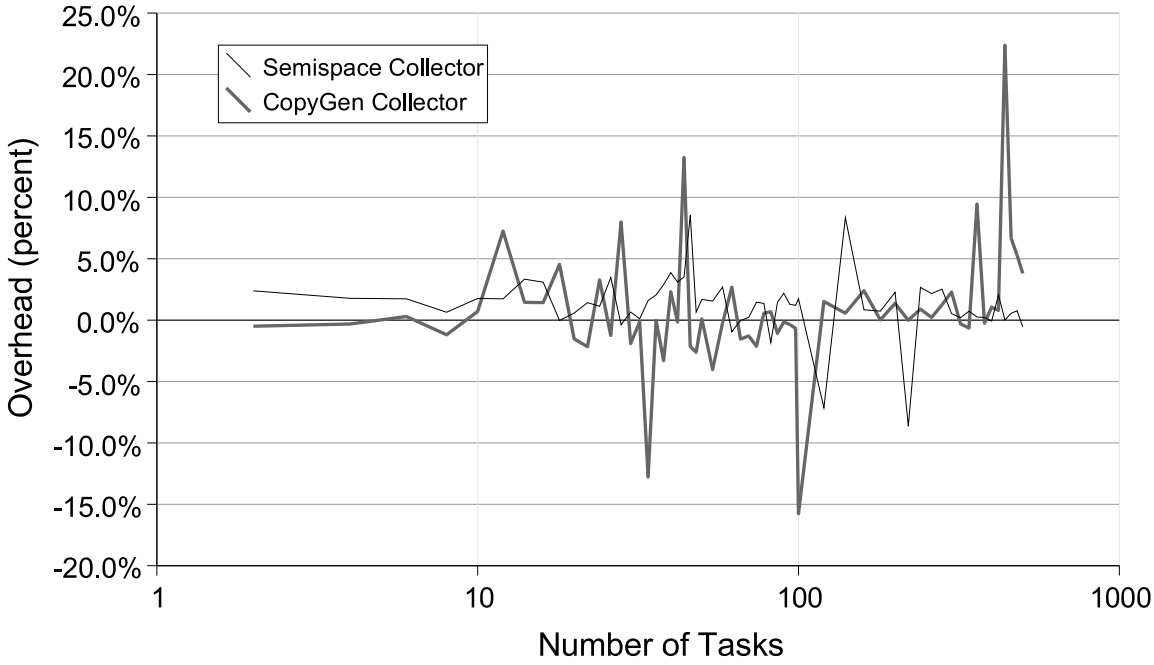

Figure 6. Runtime overhead of memory accounting on the multitasking microbenchmark with the RVM "semispace" and "copyGen" collectors, varying the number of active tasks.

ercise our system as it's intended to run. While we could have used a number of benchmarks from the database community, such as OO7 [15], these benchmarks are not primarily designed to place pressure on the garbage collector. To address this, we decided to write our own synthetic benchmark.

Our benchmark draws its inspiration from Boehm's, in that it also deals with binary trees, but in order to ensure a good degree of memory sharing, we used applicative binary trees. An applicative tree is a functional data structure, immutable once it is created. To perform an insert on an applicative tree, the node that would normally have been mutated is instead replaced with a newly allocated node, as are all of its ancestors. Each insertion thus allocates $O(\log n)$ new nodes. Throwing away the reference to the old tree likewise makes $O(\log n)$ old nodes dead, and thus eligible for garbage collection.

In our benchmark, a random applicative binary tree of tens of thousands of nodes is generated, and a reference to this tree is passed to each task. Each task then repeats two phases: adding new elements to its view of the tree, and randomly trading its view of the tree with another task's view of the tree. Each task performs a number of insertions inversely proportional to the number of tasks present in the benchmark, such that the total number of insertions performed over the benchmark run remains constant regardless of the number of tasks participating in the test. The total running time of the benchmark thus stays relatively flat as we vary the number of concurrent tasks.

We ran this benchmark using a 64MB heap size; for the generational collector, we employed a 16MB nursery heap. We measured three time values: the setup time (the amount of time required to load all of the classes and assign them to their tasks), the time spent in the garbage collector, and the remaining runtime. Our results are presented in Figure 6 and Table 1. The graph in Figure 6 shows that performance overhead varies noticeably as we increase the number of tasks, but is generally quite small. In some cases, the modified system outperformed the original system, although in other cases, the original system outperformed the modified system. Such variations most likely result from fortuitous rearrangements of how code or data collides in the processor's caches, TLB, and so forth. Table 1 shows averages over all the benchmark runs. On average, we observe that our system adds a negligible overhead to the benchmark's total running time (around 1%).

Another interesting observation is how often the garbage collector actually runs. The semispace collector runs roughly once every two seconds. With the generational collector, however, major collections tend not to occur very of-

Garbage Collector		Load Time (sec)	GC Time (sec)	Exec Time (sec)	Total Time (sec)	Major Collects	Minor Collects
Semispace	Original	0.50 ± 0.55	9.81 ± 4.25	5.50 ± 0.12	15.81 ± 4.81	7.58 ± 0.76	–
	Modified	0.47 ± 0.46	9.92 ± 4.29	5.60 ± 0.12	15.99 ± 4.78	7.60 ± 0.76	–
	Overhead	-6.14%	1.12%	1.72%	1.24%	–	–
CopyGen	Original	0.57 ± 0.56	1.82 ± 1.08	7.19 ± 0.11	9.58 ± 1.43	$0.03 \pm 0.18^{\dagger}$	13.63 ± 0.26
	Modified	0.63 ± 0.76	1.83 ± 1.08	7.21 ± 0.24	9.66 ± 1.61	$0.03 \pm 0.18^{\dagger}$	13.63 ± 0.26
	Overhead	11.19%	0.54%	0.20%	0.83%	–	–

Table 1. Mean run-time and standard deviation for the multitasking microbenchmark, across 58 runs of the benchmark, varying the number of concurrent tasks. The benchmark is run against two garbage collectors ("semispace" – a two-space copying collector, and "copyGen" – a generational collector) in two configurations ("original" – the unmodified RVM garbage collector and "modified" – adding our GC memory accounting patches). "Load time" includes the class loading and accounting system setup. "GC time" includes time spent in the GC itself, "Exec time" is the CPU time spent directly by the benchmark, and "Total time" is the sum of these components. "Major" and "Minor" are the average number of times the garbage collector was invoked during the benchmark runs. †*Of the 58 benchmark runs, there was no major collection in 56 of the runs, and exactly one major collection in the remaining two.*

ten, if at all, most likely because our benchmark is keeping relatively little data live over long periods of time, forcing any memory accounting policy to rely on data collected during the minor collectors for its policy decisions. Section 4 discusses this in more detail.

4 Discussion

The system described so far provides primitives for measuring memory being held live by various tasks in the runtime. These measurements, by themselves, do nothing; a policy engine that queries them is needed to place resource usage restrictions on tasks that are misbehaving. It's important that these policies be written with an awareness of what has actually been measured by the annotated garbage collector.

As discussed in Section 2.3.1, the usage values for a task measure two different statistics: a high-water mark, indicating how much memory that task is using, including memory it shares with other tasks, and a low-water mark that accounts for the memory used by that task alone. An intelligent policy would consider both of these values. It should be noted that these values may not reflect the most current state of how tasks are consuming memory. Tasks that have a large variation in the amount of memory allocated over time will be measured less accurately as a result. Policies might also look at intermediate measurements made when a task is neither the first nor the last to be processed by the garbage collector. These measurements may still be useful for adjusting the lower and upper-bounds on a task's memory usage.

We observe that in the presence of greater memory pressure (when it is likely to be more important to enforce limits on memory usage), the frequency of garbage collection will go up; accordingly, the frequency with which memory usage is measured also increases. This implies that the accuracy and timeliness of memory accounting will improve when it is most needed. Other factors, such as a choosing a smaller heap size, will also result in higher-resolution accounting data, although at the cost of some runtime efficiency.

When using a generational collector, the same upper and lower bounds exist. Exact measurement of memory usage is only available after a major collection. However, a generational collector is explicitly designed to minimize such collections of the mature space. For an example of the scarcity of major collections, see Table 1. However, information is available from minor collections that we can use to refine the measurements from the last major collection. The amount of memory tenured on behalf of each task during minor collections can be added to the upper bound of that task's memory usage. A similar approach is taken by MWM [21]. On the next major collection, this incremental adjustment becomes obsolete, and is discarded. As a result, the high and low boundaries measured during major collections have the same properties as the measurements made in non-generational semispace collectors.

Regardless, the quality of these measurements is not as good as the measurements available in a semispace collector. If a task is consuming an excessive amount of memory, it may take a long time to be discovered using this methodology alone. Instead, these measurements should only be

used as a way of indicating likely culprits. If the system is running low on available memory, it's unclear which task or tasks is responsible, although the larger ones observed thus far would seem likely candidates for further analysis. It's possible to explicitly force a major collection at any time, so it would be sensible in low-memory conditions to perform extra garbage collections to arrive at a culprit. Such extra analysis would make sense to avoid falsely terminating the wrong task.

This leads to a general observation: we can always trade additional CPU overhead for additional accounting accuracy. As an example, we could run a copying collector twice in succession in order to get a precise picture of some task's low and high usage statistics. The measurements that are provided by default are generated in the course of the garbage collector's normal business, but asking the garbage collector to run more often in order to improve the quality of data gathered might be a reasonable choice for some policy engines.

Other policies can be feasibly implemented beyond those based on the amount of memory being held live by each task. For example, a policy may wish to charge tasks for the time spent copying memory on their behalf by the garbage collector. With a trivial change to our annotated garbage collectors, we could push a previously hidden cost (time spent in the garbage collector) back onto the tasks that incurred it, perhaps suitably modifying the tasks' thread priorities.

Finally, we observe that accurate accounting of memory usage is predicated on there being discernible boundaries between tasks. If all tasks have references to and from some sort of central switchboard, such that every task has a path to all the memory of every other task, then the measured numbers will be out of synch with reality. Unaccountable references should be used in such systems in order to provide segmentation. For pre-existing software applications not yet adapted to use memory accounting, this may represent a non-trivial engineering effort.

5 Related work

5.1 Operating system-based resource accounting

Operating systems like UNIX have supported resource accounting and management almost since their inception. The `top` program and associated kernel facilities is a common interface for resource accounting, and the `limit` facility of the UNIX shell is the most common interface to UNIX resource management. Modern UNIX systems also include the `getrlimit(2)` and `setrlimit(2)` system calls for specifying per-process limits for a variety of system resources, including memory usage.

Several recent operating systems, including Angel [44], Opal [17] and Mungi [36], have been designed to support a single, large address space for all applications. Such systems, commonly designed for 64-bit architectures, can support data sharing semantics comparable to language-based systems, since all pointers are global. Regardless, single address space operating systems segregate memory into pages which are "owned" by and charged to specific processes, exactly as in traditional operating systems.

5.2 Language-based resource accounting

Systems such as Smalltalk [30], Pilot [45], Cedar [49], Lisp Machines [13], and Oberon [55] have taken advantage of language-based mechanisms to provide OS-like services. At least as early as the Burroughs B5000 [14] series computers, language based mechanisms were being used for security purposes. More recently, language-based enforcement of security has been popularized by Java [32, 39], originally deployed by Netscape for its Navigator 2.0 browser in 1995 to run untrusted applets.

Java popularized this approach of providing security as a side-effect of enforcing its type system. While numerous bugs have been uncovered [24, 43], significant strides have been made at understanding the type system [3, 48, 25, 26, 23, 20] and supporting expressive security policies, including restrictions that can allow trusted "system" code to run with reduced privileges [52, 31, 27, 28]. The design of Java and other multitasking run-time systems has focused primarily on type-safe security that forbids tasks from accessing data without authorization.

However, these systems provide little or no support for resource accounting and management on the programs they run. A number of projects have been developed to address this. A recent Scheme system called MrEd [29] supports thread termination and management of resources like open files. Some systems, such as PLAN [37], restrict the language to provide resource usage guarantee (termination, in this case).

Much of the recent research in this area has been focused on the Java programming language. Chander et al. [16] describe a system to target specific sorts of resource exhaustion attacks via bytecode instrumentation. The general technique they present is to replace calls to sensitive methods (for instance, for setting thread priority or creating a new window) with calls to customized methods that first verify that the operation is not harmful. Soft termination [46] is a general-purpose mechanism for terminating wayward tasks. Neither of these mechanisms address the problem of tracking resource usage, but both would be useful in conjunction with a resource accounting mechanism.

The multitasking virtual machine (MVM) [21] is a customization to the Java virtual machine implementing sepa-

ration of tasks using a process abstraction. They assign a heap to each stack; however, data that has been live for long enough is moved to a shared heap by the garbage collector. The garbage collector is used to track how much data a task has live in the shared heap. Although this approach is similar to ours, the MVM does not allow tasks to share memory; as a result, they do not address the problem of memory accounting in the face of such sharing.

Another approach that uses the garbage collector to enforce resource limitations is rent collection [4]; objects on the heap are given a store of money that is debited by the garbage collector. Objects that run out of money are "evicted" and treated as garbage. If a task is not willing or able to pay for the memory usage of its allocated objects, those objects are collected. Rent collection has also been used to prevent side-channel attacks in a multi-level system [38]. Bertino et al. also tackle the problem of designing a garbage collector for a multi-level secure heap [10, 19]. These systems are primarily concerned with covert communication channels between tasks of different security levels that are allows to share objects references; by observing when objects are or are not garbage collected, information can be covertly passed. These systems are uninterested in measuring memory usage; likewise, we do not consider garbage collector covert channels in our work.

Wick et al. [53] also present a system for using a garbage collector to determine memory usage of tasks in Scheme based on reachability. One key difference between our work and Wick et al. is that they measure only a single usage value for each task, limiting the expressiveness of their security policies when many objects are shared among tasks. They likewise have no notion comparable to unaccountable references (see Section 2.3.2) to allow one task to explicitly accept the full charges for sharing an object with another task.

5.3 Garbage collection

Garbage collection has been around since at least the LISP programming language [42]. Wilson [54] provides an excellent overview of garbage collection techniques. Some more common techniques include mark-and-sweep [41], copying collectors [18], and generational garbage collectors [50]. We implemented memory accounting for copying and generational collectors.

6 Future work

One area of future work is addressing current trends in memory management research. As noted in Section 2.3.3, porting our memory accounting system to a generational garbage collector required changes in the design of our accounting system. More recent and future advances in the state of the art of garbage collection research, such as advances in region-based memory allocation [33], will likely require similar adjustments to our memory management system.

Additionally, while we can track memory usage, there are other shared resources that are difficult to track. For instance, while accounting for CPU time spent directly by a task is straightforward, determining how much CPU time the operating system kernel has spent on behalf of each task is more complicated. Resource containers [7] offer a possible solution. Their motivation, to discover what resources are being used on behalf of some task by the kernel and charge the cost of those resources back to the task, closely parallels our own motivation to charge costs incurred by the garbage collector back onto the tasks responsible. Conceivably, a single task in the runtime could be mapped to its own resource container at the operating system level, and the resources the operating system spends on behalf of that task can be billed back to it.

The resource accounting system developed here is a measuring agent. Other systems exist to provide enforcement, limiting or terminating tasks that are deemed to be misbehaving. An interesting area for future work is flexible policy systems that read the measurements produced by this and other resource accounting systems and choose when to terminate or restrict tasks that violate the stated resource usage policy. A policy framework based on our accounting system could take into account the various statistics made available to it, like total memory copied on behalf of some task, the amount of memory that task is holding live, and the amount of memory that task is sharing with others to make its decisions.

7 Conclusion

Although Java and other general-purpose language-based systems have good support for memory protection, authorization, and access controls among mutually distrustful parties, there is little or no support for monitoring or controlling the resource usage of the individual parties. Such mechanisms allow boundaries to be established on memory usage, preventing denial of service attacks and generally increasing a system's reliability. Existing mechanisms either limit communications and memory sharing among tasks, can be fooled into charging the wrong task for memory usage, or don't gracefully support handing objects off from one task to another.

Knowing which task allocated an object is not as important as knowing which task is holding a live reference to that object. Our memory accounting system allows tasks to be charged for any memory they reference. The system is integrated into the garbage collector, piggy-backing on the periodic memory scans normally performed to main-

tain the memory heap. As a result, our system derives an accurate measure of memory usage while having almost insignificant performance overhead (typically less than 3% on a variety of benchmarks). Additionally, the accuracy of our measurements improves when there is increased memory pressure on the system, again at no additional performance cost. These measurements, combined with a suitable task termination system, allow for the graceful implementation of a variety of memory usage policies in language-based systems.

8 Acknowledgements

Ryan Culpepper, VanDung To, and Mark Barrett implemented an early prototype of this system; Ryan's advice in particular helped us get our implementation off the ground. Scott Crosby provided many useful comments and sanity checks. Thanks also to our shepherd, Drew Dean, and the anonymous reviewers for their extensive feedback.

This work is supported by NSF Grant CCR-9985332 and Texas ATP grant 003604-0053-2001.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), Feb. 2000.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[3] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Number 1523 in Lecture Notes in Computer Science. Springer-Verlag, July 1999.

[4] M. Anderson, R. D. Pose, and C. S. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, Feb. 1986.

[5] G. Back and W. Hsieh. Drawing the Red Line in Java. In *Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, Mar. 1999.

[6] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, Oct. 2000.

[7] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI)*, New Orleans, Louisiana, Feb. 1999.

[8] M. Beg and M. Dahlin. A memory accounting interface for the Java programming language. Technical Report CS-TR-01-40, University of Texas at Austin, Oct. 2001.

[9] P. Bernadat, D. Lambright, and F. Travostino. Towards a resource-safe Java for service guarantees in uncooperative environments. In *IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, Madrid, Spain, Dec. 1998.

[10] E. Bertino, L. V. Mancini, and S. Jajodia. Collecting garbage in multilevel secure object stores. In *Proceedings of the Symposium on Security and Privacy*, pages 106–120, Oakland, CA, May 1994. IEEE Computer Society Press.

[11] W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *2001 Symposium on Applications and the Internet*, San Diego, California, Jan. 2001.

[12] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, Sept. 1988.

[13] H. Bromley. *Lisp Lore: A Guide to Programming the Lisp Machine*. Kluwer Academic Publishers, 1986.

[14] Burroughs Corporation, Detroit, Michigan. *Burroughs B6500 Information Processing Systems Reference Manual*, 1969.

[15] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, 1993.

[16] A. Chander, J. C. Mitchell, and I. Shin. Mobile code security by Java bytecode instrumentation. In *2001 DARPA Information Survivability Conference & Exposition (DISCEX II)*, Anaheim, California, June 2001.

[17] 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, Nov. 1994.

[18] C. J. Chenney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, Nov. 1970.

[19] A. Chiampichetti, E. Bertino, and L. V. Mancini. Mark-and-sweep garbage collection in multilevel secure object-oriented database systems. In D. Gollmann, editor, *Proceedings of the Third European Symposium on Research in Computer Security (ESORICS)*, volume 875 of *Lecture Notes in Computer Science*, pages 359–373, Brighton, UK, Nov. 1994. Springer.

[20] A. Coglio and A. Goldberg. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience*, 13(13):1153–1171, Sept. 2001.

[21] G. Czajkowski and L. Daynès. Multi-tasking without compromise: a virtual machine approach. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, Tampa Bay, Florida, Oct. 2001.

[22] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, Vancouver, British Columbia, Oct. 1998.

[23] D. Dean. The security of static typing with dynamic linking. In *Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, Apr. 1997.

[24] D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz. Java security: Web browsers and beyond. In D. E. Denning and

P. J. Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*, pages 241–269. ACM Press, New York, New York, Oct. 1997.

[25] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, Jyväskylä, Finland, June 1997.

[26] S. Drossopoulou, D. Wragg, and S. Eisenbach. What *is* Java binary compatibility? In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 341–358, Vancouver, British Columbia, Oct. 1998.

[27] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS '98)*, pages 38–48, San Francisco, California, Nov. 1998.

[28] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, Ontario, Canada, Sept. 1999.

[29] M. Flatt, R. B. Findler, S. Krishnamurthy, and M. Felleisen. Programming languages as operating systems (*or* revenge of the son of the Lisp machine). In *Proceedings of the 1999 ACM International Conference on Functional Programming (ICFP '99)*, Paris, France, Sept. 1999.

[30] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989.

[31] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, Massachusetts, June 1999.

[32] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[33] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[34] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[35] C. Hawblitzel and T. von Eicken. Luna: a flexible Java protection system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, Massachusetts, Dec. 2002.

[36] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[37] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93, 1998.

[38] P. A. Karger. Improving security and performance for capability systems. Technical Report 149, University of Cambridge Computer Laboratory, Oct. 1988.

[39] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[40] D. Malkhi, M. Reiter, and A. Rubin. Secure execution of Java applets using a remote playground. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 40–51, Oakland, California, May 1998.

[41] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, Apr. 1960.

[42] J. McCarthy. History of LISP. In R. L. Wexelblat, editor, *History of Programming Languages*, pages 173–185. Academic Press, 1981.

[43] G. McGraw and E. W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley and Sons, New York, New York, 1999.

[44] K. Murray, A. Saulsbury, T. Stiemerling, T. Wilkinson, P. Kelly, and P. Osmon. Design and implementation of an object-orientated 64-bit single address space microkernel. In *2nd USENIX Symposium on Microkernels and other Kernel Architectures*, San Diego, California, Sept. 1993.

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

[46] A. Rudys and D. S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(2):138–168, May 2002.

[47] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, pages 202–216, Kiawah Island Resort, South Carolina, Dec. 1999.

[48] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(1):90–137, Jan. 1999.

[49] 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.

[50] D. Ungar. Generational scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Apr. 1984.

[51] L. van Doorn. A secure Java virtual machine. In *Ninth USENIX Security Symposium Proceedings*, Denver, Colorado, Aug. 2000.

[52] D. S. Wallach, E. W. Felten, and A. W. Appel. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, Oct. 2000.

[53] A. Wick, M. Flatt, and W. Hsieh. Reachability-based memory accounting. In *Third Workshop on Scheme and Functional Programming*, Pittsburgh, Pennsylvania, Oct. 2002.

[54] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, Saint-Malo, France, Sept. 1992.

[55] N. Wirth and J. Gutknecht. *Project Oberon*. ACM Press, 1992.