Garbage Collector Memory Accounting in Language-Based Systems

David Price, Algis Rudys, and 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 for accounting for heap memory usage on a per-task basis. In addition, because tasks may very well share references to the same objects, 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 on 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, or 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 that is operated on by that code.

Java [28, 34], originally supporting untrusted applets inside a web browser, popularized this approach of providing security as a side-effect of enforcing its type system. While numerous bugs have been uncovered [20, 38], significant strides have been made at understanding the type system [2, 43, 21, 22, 19, 16], and supporting expressive security policies, including restrictions that can allow trusted “system” code to run with reduced privileges [47, 27, 23, 24]. However, language run-time systems do not have all the same protection semantics as operating system processes. Processes encapsulate all the memory being used by a given task, making it easy to measure the total memory in use and 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, process-structured systems limit the ability
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 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.

This paper describes a new mechanism for tracking memory usage of tasks. Language-based systems use garbage collectors to provide memory management services. These garbage collectors are already empowered to examine the entire heap and determine which memory is used by the systems. 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.

We also strive to provide sufficient information for a flexible policy language. The system maintains statistics not only on the amount of memory a task uses, but also 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 [41] 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 [18] and Beg and Dahlin [6] 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, accounting is performed on a per-thread basis. If a “system” thread or another task’s thread calls into a task, it could potentially be “tricked” into allocating memory on behalf of the task which is not charged to that task.

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 exist to it. 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 [3, 4] 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) [17] and systems by Bernadat et al. [7] and van Doorn [46] similarly use separate heaps or memory spaces to facilitate accounting for memory. Some systems [35, 42] 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 gives an accurate number for the amount of memory used by a
task. Examples include J-Kernel [30], J-SEAL2 [8], and Luna [31]. 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 being used by the 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, doing a graph traversal to find all the memory reachable from those references. Memory not discovered during this graph traversal is unreachable, and therefore can be considered “dead” and thus safe to reuse for 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, using the “normal” root set, that would reach all live objects on the heap. Any objects remaining after this completes are considered to be 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 task processed shares with others, and 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.

We solve the difficulty by rotating the order that tasks are processed on subsequent collections, so that there’s a different one processed first each time. The effect of this can be seen by comparing Figures 3 and 4; changing the processing order changes the memory found for each task.

The first task processed yields a maximum value—an upper bound on memory usage including memory that’s shared. Processing the last one in line yields a lower bound, indicating how much memory that task is responsible for that no other task has a reference to. Results in the middle give an intermediate picture 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 good approximation for how much memory each task is sharing. The issue of synthesizing this raw information into useful policies is discussed in Section 4.
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).
2.3.2 Unaccountable references

One concern of garbage collector-based memory accounting has been described as the “resource Trojan horse” problem [31]. 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 some untrusted code without exposing itself to such an attack. Similarly, a system library providing access to a database may (generously) not want the task to be charged for storage of that 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 naively 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. The result is that if the only path a task has to some memory is through an unaccountable reference, that memory will be guaranteed to be held live, but that task will not be charged for that memory.

However, 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.

Another possible approach is to permit any task to create an unaccountable reference, but 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 sort of technique, which is implemented as a slight tweak 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 trace 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.
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 [9] 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 this 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 and 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”). GCTk1 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 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 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 512 MB 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 benchmark2, 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.

---

1 See http://www.cs.umass.edu/~gctk/.
Figure 5: Percentage of overhead incurred by the accounting modifications on Boehm’s artificial GC benchmark and on various real-world application benchmarks.

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, a LALR parser-generator, Linpack, an implementation of a matrix multiply, and OTP, 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 them, so it’s not surprising for our changes to the GC system to 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 each other, traditional single-tasking benchmarks are insufficient to exercise our system as it’s intended to run. While we could have used a number of benchmarks from the database community, such as OO7 [12], 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 amount of tasks participating in the test. The total running time of the benchmark thus stays relatively flat, regardless of the number of concurrent tasks executing.

We ran this benchmark using a 64 MB heap size; for the generational collector, we employed a 16 MB 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 figure 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, where 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. The table 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 often, if at all, most likely because our benchmark is keeping relatively little data live over long periods of time, forcing a memory accounting policy to rely on data collected during the minor collectors for its policy decisions. Section 4 discusses this in more detail.

Figure 6: The overhead of memory accounting on the multitasking microbenchmark with the RVM “semispace” and “copyGen” collectors, varying the number of active tasks.
In 56 of the runs, and exactly one major collection in the remaining two.

By the accounting system. Other factors, like a smaller choice of heap size, will also result in higher-
accounting: the more quickly tasks allocate memory, the more rapidly their memory usage will be discovered
with which memory usage is assessed also increases. Effectively, this translates to a sort of on-demand
place resource usage restrictions on tasks that are misbehaving. It’s important that these policies be written
with an eye towards the properties of the measurements.

As discussed in Section 2.3.1, the usage counts revealed for a task track two different statistics: a high-
water mark, indicating how much memory that task is using, including that which it shares with others,
and a low-water mark that accounts for the memory that just that task is using. An intelligent policy would
consider both of these numbers. It should be noted that these water marks are not necessarily updated as of
the last garbage collection. 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 set
an upper and lower bound on the low and high water marks, respectively.

We do observe that in the presence of greater memory pressure (when it is likely more important to
enforce limits on memory usage), the frequency of garbage collection will go up; accordingly, the frequency
with which memory usage is assessed also increases. Effectively, this translates to a sort of on-demand
accounting: the more quickly tasks allocate memory, the more rapidly their memory usage will be discovered
by the accounting system. Other factors, like a smaller choice of heap size, will also result in higher-
resolution accounting data, at the cost of some runtime efficiency.

In the presence of a generational collector, the same upper and lower bound figures exist, but are only
updated with each major collection. Since a generational collector is explicitly designed to minimize collec-
tions of the mature space, these collections are likely to be few and far between (for an example of the
scarcity of major collections, see Table 1.) We partially solve the problem by counting the amount of mem-
ory tenured on behalf of each task in between minor collections; this gives an idea of how much memory
in the mature heap is being used by each task. This approach of incrementing a counter upon tenuring is
similar to the one MVM [17] takes in tracking memory usage of tasks.

Termination policies might not want to just look at the numbers as generated by the generational collec-

<table>
<thead>
<tr>
<th>Garbage Collector</th>
<th>Load Time (sec)</th>
<th>GC Time (sec)</th>
<th>Exec Time (sec)</th>
<th>Total Time (sec)</th>
<th>Major Collects</th>
<th>Minor Collects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semispace</td>
<td>Original</td>
<td>0.50 ± 0.55</td>
<td>9.81 ± 4.25</td>
<td>5.50 ± 0.12</td>
<td>15.81 ± 4.81</td>
<td>7.58 ± 0.76</td>
</tr>
<tr>
<td></td>
<td>Modified</td>
<td>0.47 ± 0.46</td>
<td>9.92 ± 4.29</td>
<td>5.60 ± 0.12</td>
<td>15.99 ± 4.78</td>
<td>7.60 ± 0.76</td>
</tr>
<tr>
<td></td>
<td>Overhead</td>
<td>−6.14%</td>
<td>1.12%</td>
<td>1.72%</td>
<td>1.24%</td>
<td>−</td>
</tr>
<tr>
<td>CopyGen</td>
<td>Original</td>
<td>0.57 ± 0.56</td>
<td>1.82 ± 1.08</td>
<td>7.19 ± 0.11</td>
<td>9.58 ± 1.43</td>
<td>0.03 ± 0.18</td>
</tr>
<tr>
<td></td>
<td>Modified</td>
<td>0.63 ± 0.76</td>
<td>1.83 ± 1.08</td>
<td>7.21 ± 0.24</td>
<td>9.66 ± 1.61</td>
<td>0.03 ± 0.18†</td>
</tr>
<tr>
<td></td>
<td>Overhead</td>
<td>11.19%</td>
<td>0.54%</td>
<td>0.20%</td>
<td>0.83%</td>
<td>−</td>
</tr>
</tbody>
</table>

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 config-
urations (“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” in-
cludes 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.

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 eye towards the properties of the measurements.

We observe that in the presence of greater memory pressure (when it is likely more important to
enforce limits on memory usage), the frequency of garbage collection will go up; accordingly, the frequency
with which memory usage is assessed also increases. Effectively, this translates to a sort of on-demand
accounting: the more quickly tasks allocate memory, the more rapidly their memory usage will be discovered
by the accounting system. Other factors, like a smaller choice of heap size, will also result in higher-
resolution accounting data, at the cost of some runtime efficiency.

In the presence of a generational collector, the same upper and lower bound figures exist, but are only
updated with each major collection. Since a generational collector is explicitly designed to minimize collec-
tions of the mature space, these collections are likely to be few and far between (for an example of the
scarcity of major collections, see Table 1.) We partially solve the problem by counting the amount of mem-
ory tenured on behalf of each task in between minor collections; this gives an idea of how much memory
in the mature heap is being used by each task. This approach of incrementing a counter upon tenuring is
similar to the one MVM [17] takes in tracking memory usage of tasks.

Termination policies might not want to just look at the numbers as generated by the generational collec-
tor’s accounting features in order to make their decisions. It’s possible to explicitly force a major collection at any time; a policy engine might do so when the numbers it sees give it a hint that some task might be overstepping its memory usage bounds. Spending the extra time to do the major collection could give it the numbers it needs to make a definite decision as to whether to terminate or otherwise restrict the task in question.

The above idea leads to a general observation: it’s easy to throw more computing resources at the problem of accounting in order to get more accurate information. For instance, we could run a copying collector twice in succession in order to get a precise picture of some task’s low and high water mark figures. The measurements that are provided by default are provided 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.

We can envision other policies besides those that produce a count of memory being held live. For instance, it makes sense to bill tasks for the total amount of memory copied on their behalf by the collector. The result of such a policy would be to push a previously hidden cost (time spent in the garbage collector) back onto the tasks that incurred it.

Finally, we observe that accurate accounting of memory usage is predicated on there being some 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.

5 Related work

5.1 Operating System-Based Resource Accounting

Our system seeks to use a language run-time system’s garbage collector to provide operating system-style memory accounting to the language run-time system. Operating systems like UNIX have supported resource accounting almost since their inception. The limit facility of the UNIX shell is the most common interface to UNIX resource accounting. 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.

Recent research has also been pursued on operating systems that run all their applications in a single address space. Such systems, commonly designed for 64-bit architectures, generally support similar data sharing semantics to those of language-based systems. Most single address space operating systems provide memory accounting semantics similar to traditional operating systems: a single entity is charged for each memory region in use, and is free to deallocate the region at will. Angel [39], Opal [14], and Mungi [32] are example single address space operating systems.

5.2 Language-Based Resource Accounting

Systems such as Smalltalk [26], Pilot [40], Cedar [44], Lisp Machines [10], and Oberon [49] have taken advantage of language-based mechanisms to provide OS-like services. At least as early as the Burroughs B5000 [11] series computers, language based mechanisms were being used for security purposes. More recently, language-based enforcement of security has been popularized by Java, originally deployed by Netscape for its Navigator 2.0 browser in 1995 to run untrusted applets.

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 [25] supports thread termination and management of resources like open files. Some systems, such as PLAN [33], 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. [13] 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 [41] is a general-purpose mechanism for terminating wayward tasks. Neither of these mechanisms address the problem of tracking resource usage, but both could be used in conjunction with a resource accounting mechanism.

The multitasking virtual machine (MVM) [17] is a customization to the Java virtual machine implementing separation 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, it does not attempt to account for memory usage in the face of such sharing.

5.3 Garbage Collection

Garbage collection has been around since at least the LISP programming language [37]. Wilson [48] provides an excellent overview of garbage collection techniques. Some of the more common garbage collection techniques include mark-and-sweep [36], copying collectors [15], and generational garbage collectors [45]. 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 [29], 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 [5] 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 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 stated resource usage policy. A policy framework built to this 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 mechanism is useful for the purpose of quotas or preventing denial of service attacks. 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 task to another.

In this project, we recognize that knowing which task allocated an object is not as important as knowing which task is preventing an object from being deallocated. Our memory accounting system charges tasks for any memory the task contributes to keeping alive. The system is integrated into the garbage collector, which already scans the memory on a periodic basis in managing a language-based system’s heap. As a result, our system gets an accurate count of memory usage not possible through simply instrumenting explicit allocations, while avoiding the additional overhead of scanning the system’s heap. There is an additional advantage that when there is memory pressure and memory accounting is most important, the garbage collector is run more frequently and the accounting figures are most accurate.

Our system has an overhead of less that 3% in application benchmarks. The resulting system is fast, does not interfere with the garbage collector, and integrates very cleanly into a language’s memory management system.

Acknowledgements

Ryan Culpepper, VanDung To, and Mark Barrett implemented an early prototype of this system. The authors would like to thank Scott Crosby and Tsuen “Johnny” Ngan for their helpful suggestions. Robert “Corky” Cartwright also provided some helpful advice, for which the authors are grateful.

References


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


