Memory Accounting Without Partitions

Adam Wick
awick@cs.utah.edu

Matthew Flatt
mflatt@cs.utah.edu

University of Utah, School of Computing
50 South Central Campus Drive, Room 3190
Salt Lake City, Utah 84112–9205

ABSTRACT

Operating systems account for memory consumption and allow for termination at the level of individual processes. As a result, if one process consumes too much memory, it can be terminated without damaging the rest of the system. This same capability can be useful within a single application that encompasses subtasks. An individual task may go wrong either because the task's code is untrusted or because the task's input is untrusted. Conventional accounting mechanisms, however, needlessly complicate communication among tasks by partitioning their object spaces. In this paper, we show how to provide applications with per-task memory accounting without per-task object partitions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features–*Dynamic Storage Management*;
D.3.4 [**Programming Languages**]: Processors–*Run-time environments*

General Terms

Languages, Reliability

Keywords

Garbage collection, memory accounting, concurrent programming, software reliability

1. INTRODUCTION

As applications grow increasingly complex, they are increasingly organized into smaller subprograms. For example, web browsers invoke external programs to display images and movies, and spreadsheets frequently execute user-defined scripts. The more subtasks that an application invokes the more things can go wrong, and the more it becomes useful to control the subtasks. In this paper, we concentrate on the problem of constraining memory use.

ISMM'04 October 24–25, Vancouver, British Columbia, Canada.

Applications currently restrict memory use by partitioning data and then limiting the memory use of the partitions. Traditional operating systems partition memory into completely separate heaps for each process, disallowing references between them. This strict partitioning makes interprocess communication difficult, requiring the marshaling and demarshaling of data through pipes, sockets, channels or similar structures. In some cases, marshaling important data proves infeasible, leading to the use of complicated protocol programming.

More recent work provides hard resource boundaries within a single virtual machine. Systems in this class, such as the KaffeOS virtual machine [2], JSR-121 [13] or .NET application domains [11], still partition data, but without the separate address space. Generally, the programmer explicitly creates a shared partition and may freely allocate and reference objects in it. However, these systems do place restrictions on inter-partition references. For example, KaffeOS disallows references from its shared heap to private heap space. This less strict form of partitioning only partially alleviates the burden on the programmer. While the program may now pass shared values via simple references, the programmer must explicitly manage the shared region. In the case where one process wants to transfer data completely to another process, the transfer may require two deep copies: one to transfer the data into shared space, and the other to transfer it out. In short, the programmer must manually manage accounting in much the same way a C programmer manages memory with $malloc()$ and $free()$.

Our system of *partition-free* memory accounting provides the accounting ability of partitions without unnecessary work by programmers. Further, as a *consumer-based* system, programmers simply allocate and pass objects around as they please, and the current holder of the object is charged, rather than the allocator. Thus, data migration and sharing require no additional work on the part of the programmer: no marshaling, no complex communications, and no explicit management of partitions. By leveraging an existing garbage collector and the process hierarchy [10], our system is flexible enough to handle most memory accounting demands, and is fast enough to be used in production quality software. Finally, our system exports simple but reliable guarantees, which are easy for the programmer to reason about.

Although Price et al. [12] address memory accounting in a way similar to our system, they do not build on a process hierarchy, which is a cornerstone of our work. We believe that a consumer-based accounting mechanism must be tied to a process hierarchy to provide useful guarantees to

the programmer. Our practical applications–as well as our comparison to other accounting mechanisms—both depend crucially on parent–child relationships among processes.

We begin our discussion in section 2 with example applications where partition-free, consumer-based memory accounting saves valuable programmer time and energy. We follow the examples with a brief overview of our core system, MzScheme, in section 3, and we present the details of our accounting system in section 4. Section 5 describes how we use this system in the examples in section 2. Section 6 outlines how our system works over large, general classes of applications. We discuss the implementation of the system in section 7, including an analysis of the cost of accounting. Finally, we discuss related work in section 8 and conclude in section 9.

2. MOTIVATING APPLICATIONS

Conventional partitioning or producer-based accounting techniques can be used to solve most accounting problems. However, these techniques often make programming difficult for no reason. We present three applications in this section where using consumer-based, partition-free memory accounting greatly simplifies our implementation task.

2.1 DrScheme

The DrScheme programming environment consists of one or more windows, each split into a top and bottom half. The top half provides standard program editing tools to the user, while the bottom half presents an interactive Scheme interpreter. Normally, users edit a program in the top half and then test in the bottom half.

As a development environment, DrScheme frequently executes incorrect user code, which must not make DrScheme crash. Language safety protects DrScheme from many classes of errors, but a user program can also consume all available memory. Thus, DrScheme must account for memory used by the user program, and DrScheme must terminate the program if it uses too much.

As we have noted, the usual way to account for memory use and enforce memory limits is to run the interpreter in a completely separate heap space. However, this separation requires significant programmer effort. DrScheme would have to communicate with its child processes through pseudovalues rather than actual values and function calls, much like a UNIX kernel communicates with processes through file descriptors rather than actual file handles. This approach becomes particularly difficult when writing DrScheme tools that interact with both the program in the editor and the state of the interpreter loop, such as debuggers and profilers.

DrScheme could partition values without resorting to separate heaps, but partitioning only solves some problems. The programmer can either keep the entire system within one partition, or split the system into many partitions. Neither approach works well in practice:

- Allocating all objects into a single, shared partition makes communication simple, as references may be passed freely. However, DrScheme frequently invokes two or more interpreter loops. Since every object resides in the same partition, the accounting system could give no information about which interpreter DrScheme should kill if too many objects are allocated.
- Separating DrScheme into several partitions leaves no natural place for shared libraries. Shared libraries would either need to be duplicated for every partition, or must be told into which partition to allocate at any given time. The first – duplication – has obvious problems. The second amounts to manual memory management, requiring unnecessary time and effort on the part of the programmer.

A producer-based accounting scheme suffer from similar problems. In order to write simple APIs, the program in test often invokes parts of DrScheme to perform complex operations. Some of these operations allocate data, and then return this data to the child process. In this case, a producer-based system would either not honor the data hand-off or would require the parent process to communicate to the accounting system that it is allocating the data on behalf of another process. In the latter case, security becomes an additional problem.

Instead of developing complex protocols to deal with separate address spaces or partitions, our system allows a direct style of programming, where a small addition to DrScheme provides safety from overallocating programs.

2.2 Assignment Hand-In Server

Students in CS2010 at the University of Utah submit homework via a handin server. The server then tests the student's program against a series of teacher-defined tests.

This server clearly requires memory constraints. First, an incorrect program that allocates too much memory on a test input may kill the entire handin server. Second, and unlike DrScheme itself, a malicious student might attempt to bring down the server by intentionally writing a program that allocates too much memory.

Running the student program interpreter in the same process as the server saves a great deal of programming work, and saves the course instructor time by not requiring test case input and results to be marshaled. Further, we avoid problems duplicating and reloading libraries. Duplication, for example, creates a problem for test cases that use library data types, since the types in a student program would not match test cases generated by the testing process. Reloading becomes a further problem for advanced student projects, which typically require large support libraries. Reloading these libraries for every test may stress the CPU beyond acceptable levels, particularly around assignment due dates.

Partitioned accounting schemes solve some of these problems, but not all of them. As in DrScheme, shared libraries are problematic, requiring either a great loss of information or a form of manual memory management. Again, as in DrScheme, producer-based accounting schemes fail when shared libraries allocate memory on behalf of a student program.

Instead of carefully managing such details, our server uses partition-free, consumer-based accounting and works with no special protocols. The server requires neither copying code nor protocol code, and shared libraries are loaded once for the entire system.

2.3 SirMail

SirMail began as a modest mail client, and was gradually extended with a few additional features, including HTML rendering and attachment processing. These two extensions, however, introduce a memory security problem.

The HTML engine renders HTML email in the normal way. In doing so, however, it may download images from unknown, untrusted servers. By creating a small email which requires the display of an arbitrarily large graphic, an attacker (or spammer) could easily bring the whole system to a grinding halt. Attachment processing also presents a problem, as a huge attachment may arbitrarily pause the whole mail client for the duration of processing.

Partitioned accounting systems solve both these problems but create new ones. In order to function, the HTML rendering engine must draw to the main SirMail window. Similarly, SirMail must pass the data to be processed to the attachment processor, and then receive the result. Partitioning schemes cause problems for both interactions. Since shared memory regions may not contain pointers to private memory regions, in order for the HTML engine to draw to the screen either SirMail must place the entire GUI (and all callbacks and other references) into a shared region or the engine must communicate to SirMail using some drawing protocol. In the case of the attachment processor, SirMail would have to copy the entire attachment into and then out of the subprocess's heap for accounting to make sense.

Using partition-free accounting solves both these problems. SirMail simply passes a direct reference to the GUI to the rendering engine, which can then interact with it in the usual ways. Similarly, SirMail may simply call the attachment processor with a reference to the data, and have a reference to the result returned to it, requiring no copying.

3. PROCESSES IN MZSCHEME

Our base system, MzScheme [9], exports no single construct that exactly matches a conventional process. Rather, various orthogonal constructs implement different aspects of processes to give the programmer a finer grain of control over programs. While MzScheme supplies many different such constructs, only three are relevant to the current work:

- *Threads* implement the execution aspect of a process. The MzScheme *thread* function consumes a thunk and runs the thunk in a new thread, as follows:

```
(letrec ([loop (lambda (v)
                 (display v)
                 (loop v))])
  (thread (lambda () (loop 1)))
  (loop 2))
```

  In this example, the body of the **letrec** creates a thread that prints the number 1 endlessly while the rest of the body prints the number 2 endlessly.

- *Parameters* implement process-specific settings, such as the current working directory. MzScheme represents parameters with functions for getting and setting the parameter value. These values are thread-local; a newly created thread will inherit its initial value from the current value in the creating thread. Afterwards, modifying the value in the new thread has no effect on the parent thread's value.

  The following example sets the current directory to `"/tmp"` while running *do-work*, then restores the current directory:[1]

```
(let ([orig-dir (current-directory)])
  (current-directory "/tmp")
  (do-work)
  (current-directory orig-dir))
```

- *Custodians* implement the resource-management aspect of a process. Whenever a thread object is created, port object opened, GUI object displayed, or network-listener object started, the object is assigned to the current custodian, which is determined by the *current-custodian* parameter. The main operation on a custodian is *custodian-shutdown-all*, which terminates all of the custodian's threads, closes all of its ports, and so on. In addition, every new custodian created with *make-custodian* is created as a child of the current custodian. Shutting down a custodian also shuts down all of its child custodians.

  The following example runs *child-work-thunk* in its own thread, then terminates the thread after one second (also shutting down any other resources used by the child thread):

```
(let ([child-custodian (make-custodian)]
      [parent-custodian (current-custodian)])
  (current-custodian child-custodian)
  (thread child-work-thunk)
  (current-custodian parent-custodian)
  (sleep 1)
  (custodian-shutdown-all child-custodian))
```

  A thread's current custodian is *not* the same as the custodian that manages the thread. The latter is determined permanently when the thread is created.

Threads and custodians form the basis of our accounting system. A thread necessarily references its continuation and parameter values, which provides an intuitive framework for determining what objects a particular thread uses. Custodians naturally serve as resource-management objects, and provide a safe way to terminate a set of threads and all their open ports, sockets, GUI objects, etc. Therefore, our API provides memory restricting functions that work at the granularity of a custodian.

Custodians further provide an ideal granularity due to their hierarchical nature. *Custodian-shutdown-all* not only terminates threads and closes ports, it also shuts down any custodians created as a child of the given custodian. This hierarchy provides an immediate solution for an obvious security flaw, wherein a malicious custodian creates a subcustodian in order to have the subcustodian killed off instead of it. The attack simply fails, since the parent of the child custodian remains responsible for the child's memory. Furthermore, custodians are garbage collected (and thus memory accounted) entities, so a malicious child process cannot starve the system for memory simply by repeatedly allocating new custodians.

4. CONSUMER-BASED ACCOUNTING

Without sharing, accounting is a simple extension of garbage collection. Sharing complicates matters, since the system must provide guarantees as to which process or processes the shared objects will be accounted. In other words, if custodian A and custodian B both share a reference to x, the

[1]Production code would use the *parameterize* form so that the directory is restored if *do-work* raises an exception.

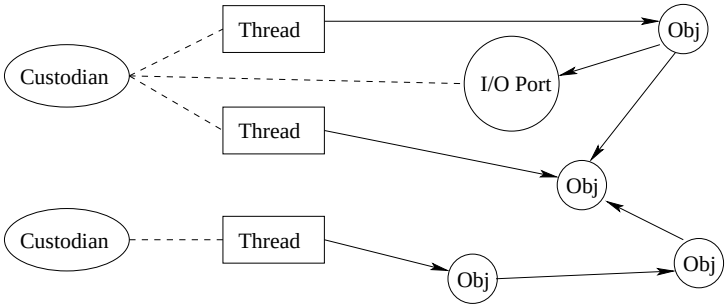

Figure 1: Custodians and threads. Threads implement concurrent execution, and hold and use objects. Custodians manage threads, along with other resources like I/O ports

accounting system must provide some guarantee as to how the charge for x will be assigned.

Certain policy decisions would provide useful guarantees, but drastically reduce performance. One such policy is to charge the use of x to both A and B, and another might split the charge equally between A and B. While these policies provide reliable, intuitive guarantees, they both suffer from a performance problem: a worst-case execution time of $C*R$, where C is the number of custodians and R is the number of reachable objects. Our experience suggests that this extra factor of C scales poorly in real systems, and provides a new opportunity for denial of service attacks (i.e., by creating many custodians).

Our approach makes one simple guarantee to the programmer that nevertheless provides useful information. We guarantee the charge of x to A, and not B, if A descends (hierarchically) from B. Due to the hierarchical nature of custodians, these charges bubble up to the parents, regardless, so assigning the charge to the child provides more useful accounting information. In other words, since B is responsible for the behavior of A, B eventually becomes charged for A's usage. In the case of unrelated sharing custodians, the charge is assigned arbitrarily. For reasons that we discuss at length in the following section, we have not found the arbitrary assignment a problem. In fact, we find that this simple approach applies well to a large class of potential applications.

Finally, the policy must describe what it means for a custodian to reference an object. We consider an object reachable by a custodian if it is reachable from any thread that custodian manages, with a few exceptions. First, since weak references do not cause an object to be retained past garbage collection, a custodian holding an object only through a weak reference is not charged for it. Second, many threads hold references to other threads and custodians. However, these references are opaque, so the original thread cannot then reference anything in that thread or custodian. Therefore, if an object x is reachable by custodian C only through a reference to some other thread or custodian, then C is not charged for x.

Given the above guarantees and policies, we export the accounting API as follows:

- (*custodian-limit-memory cust1 limit-k cust2*) installs a limit of *limit-k* bytes on the memory charged to the custodian *cust1*. If ever *cust1* uses more than *limit-k* bytes, then *cust2* is shut down.

  Typically, *cust1* and *cust2* are the same custodian, and the parent custodian uses the child custodian in question for both arguments. Distinguishing the accounting center from the cost center, however, can be useful when *cust1* is the parent of *cust2* or vice-versa.

Although *custodian-limit-memory* is useful in simple settings, it does not compose well. For example, if a parent process has 100 MB to work with and its child processes typically use 1 MB but sometimes 20 MB, should the parent limit itself to the worst case by running at most 5 children? And how does the parent know that it has 100 MB to work with in the case of parent-siblings with varying memory consumption?

In order to address the needs of a parent more directly and in a more easily composed form, we introduce a second interface:

- (*custodian-require-memory cust1 need-k cust2*) installs a request for *need-k* bytes to be available for custodian *cust1*. If *cust1* is ever unable to allocate *need-k* bytes (if it suddenly needed this amount), then *cust2* is shut down.

Using *custodian-require-memory*, a parent process can declare a safety cushion for its own operation but otherwise allow each child process to consume as much memory as is available. A parent can also combine *custodian-require-memory* and *custodian-limit-memory* to declare its own cushion and also prevent children from using more than 20 MB without limiting the total number of children to 5.

All of the above procedures register constraints with the accounting system separately. Thus, a child processes cannot raise a limit on itself by simply reinvoking *custodian-limit-memory* as both limits remain in effect. Furthermore, note that killing a custodian simply closes all its ports, stops all its threads, and so on, but does not explicitly deallocate memory. Because of this, memory shared between a custodian being shutdown a surviving custodian will not be deallocated by the shutdown process.

In addition to the two memory-monitoring procedures, MzScheme provides a function that reports a given custodian's current charges:

- (*current-memory-use cust*) returns the number of allocated bytes currently charged to custodian *cust*.

These procedures, in combination, provide simple mechanisms for constraining the memory use of subprocesses. In

most cases, extending an existing application to use memory constraints requires only a few additional lines of code. For example, consider the following program:

```
(define (simple-test-server is-ok?)
  (run-server 4343
    (lambda (in-port out-port)
      (let ([student-program (read in-port)])
        (write (is-ok? student-program)
               out-port)))
    1000))
```

This code implements a simple handin server, using the library function *run-server*. A student connects to port 4343 and sends a program. After the server completes the connection, *run-server* spawns the given function in a separate thread and custodian, and reads in the program. It then sends back, through the output port, the results of testing the program. Using the custodian, the *run-server* function imposes a timeout of 1000 seconds on this computation.

Of course, neither timeouts nor language safety protects the server itself from a memory attack. If the *is-ok?* function runs the student's code, and if the code allocates too much memory, the whole system will halt. Worse, if the client simply sends a huge program, reading in the program in the first place will crash the system.

To solve the problem we modify the program to use memory constraints, as follows:

```
(define (simple-test-server is-ok?)
  (run-server 4343
    (lambda (in-port out-port)
      (custodian-limit-memory (current-custodian)
        (* 32 1024 1024) (current-custodian))
      (let ([student-program (read in-port)])
        (write (is-ok? student-program)
               out-port)))
    1000))
```

In short, the addition of one function call prevents a bad or huge program from crashing the server.

5. ACCOUNTING IN THE EXAMPLES

Using our new accounting mechanisms, we easily solved the accounting problems described in section 2. In this section, we report briefly on our experience.

5.1 DrScheme

In order to support a stop (break) button, DrScheme runs every interpreter window in a separate custodian. These custodians descend from a single, parent custodian for the system. Since our accounting mechanism charges shared memory to the child, rather than the parent, none of our existing interfaces required updating. DrScheme simply allocates complex objects and transfers them directly to the child. Additionally, the DrScheme process has direct access to the closures, environment, and other interpreter state of a user program.

Our initial pass to add accounting to DrScheme required only four lines of code. Specifically, the four lines locally bind the new custodian, set the memory limit, and proceeded as normal.

However, a problem quickly became evident. The system, as originally constructed, contained reference links from the child custodian to the parent (through ports, GUI objects and so forth passed down through the API), but also links from the parent down to each of the children. The accounting pass, then, picked one of the children to account to first, and that child was charged for most of the objects allocated in the entire system. Since the child can reach up to the parent's object space and then back down to all its siblings' object spaces, it can reach (and thus is charged for) all of these objects.

Initially, we attempted to break all the links from the child to the parent. However, doing so creates many of the same problems as the old operating system process solution. For example, instead of handing out I/O ports directly, a file handle system must be used.

Rather than rewriting a huge chunk of DrScheme, we investigated breaking the links from the parent to the child. This modification turned out to be quite simple, involving only a few hours of programmer time to find the links from parent to child, plus another half hour to remove these links. In all, the task required the changing of five references: two were changed to weak links, one was pushed down into the child space and the final two were extraneous and simply removed.

5.2 Hand-In Server

To make the hand-in server shut down overconsuming test custodians, a single line was needed. We also added a feature to report to students when the test was terminated. Even this extra feature proved fairly simple, with the entire change comprising around 25 lines.

5.3 SirMail

Modifying the existing code to limit the memory use of the HTML rendering engine required about 45 minutes of time from a programmer unfamiliar with the SirMail code base, and about 20 lines of code. Most of this additional code detects and reports when the accounting system shuts down the rendering custodian.

The MIME processing modifications turned out to be easier, requiring approximately 10 minutes of time and an additional 5 lines of code. These five lines implement a pattern for libraries and callbacks that is described in section 6.2.1.

6. ACCOUNTING PARADIGMS

In this section, we describe several common paradigms for multiprocess programs, and show how our system easily handles most of them. Despite our apparently weak guarantee for shared-object accounting, in many cases our accounting system improves on existing mechanisms.

6.1 Concurrent Paradigms

We first discuss accounting paradigms involving multiple, communicating processes. Figure 2 gives a pictorial representation of the three possible communication paradigms.

6.1.1 Noncommunicating Processes

In some cases, a program must offload some work and does not care about the results. In such examples, the parent spawns the child with some initial data set, and then largely ignores the child unless it raises an error. Examples include print spoolers, nonquerying database transactions (inserts, updates, etc.), and logging utilities. This protocol roughly matches traditional operating system processes.

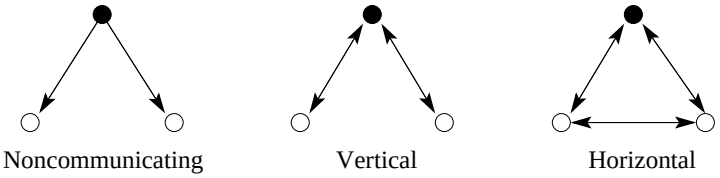

Figure 2: The three interprocess communication patterns. The filled circle represents the parent process, with the hollow circles representing child processes. The arrows represent directions of communication.

Conventional process accounting suffices in such tasks, due to the small amount of one-way communication between the two processes, but our system works equally well. Since the data in the subprocess never escapes that subprocess, any data that the subprocess uses is charged to it.

6.1.2 Vertically Communicating Processes

Another common paradigm involves two-way communication between a parent process and a child process, but not between child processes. Examples include web browsers, programming language IDEs, file processors, database queries and so on. In these cases, the parent process may create an arbitrary number of children, but these children communicate only with the parent and not each other.

Such purely vertical communication paths represent a large subset of concurrent, communicating programs involving untrusted processes. Generally, the parent program runs the untrusted process and communicates with it directly. Communication between other subprocesses and the untrusted process usually pass through the parent program. Meanwhile, the parent and each child must cooperate closely, and this cooperation is most easily implemented by sharing closures and objects.

Our algorithm for accounting memory clearly covers this case. As our accounting mechanism always accounts shared objects to the child, information on the memory usage of the child remains exact. Thus, by leveraging the custodian hierarchy, we create a framework providing exactly what the programmer wants and needs. In general, we find our system allows applications in this class to restrict the memory use of their subprocesses with little to no impact on the way they allocate and share data.

6.1.3 Horizontally Communicating Processes

Sometimes, a parent spawns multiple children that communicate directly. Examples include AI blackboard systems and parallel sorting algorithms. In such cases, the children work collaboratively to solve some problem.

When two sibling processes share data, our algorithm guarantees only that one will be charged, but does not guarantee which. Therefore, on one garbage collection, assignment of the charge may go to one process and on some other during the next. This uncertainty in charging reflects that, typically, individual charges make little sense and that killing one process will not allow others to continue. In that case, children can be grouped under a grouping custodian, which lies between the parent custodian and the child custodians in the hierarchy. The parent then sets limits on the set of children, and shuts them down as a group if they violate any memory constraints.

Another possibility is that the children may be working on disjoint sets of data, so charges make sense for each process and, presumably, individual children can proceed even if others die. In these cases, a limit on the children as a group makes little sense. However, the parent may set a memory requirement, to guarantee that a certain amount of memory is held in reserve for itself. When a violation of this requirement occurs, the program simply kills off the subprocesses in some static or heuristic order.

6.2 Single-Process Paradigms

The three previous sections concentrate on concurrent subprocesses, where consumer-based accounting makes sense. In some cases, this assumption does not apply. The first case that we consider involves untrusted libraries or callbacks installed by untrusted code. The second case involves situations (concurrent or not), where the application requires producer-based accounting.

6.2.1 Libraries and Callbacks

Some applications link (statically or dynamically) to untrusted libraries. Further, some concurrent applications install callbacks from child processes into the parent process. In these cases, the desired granularity for accounting more closely resembles a function call than a thread or process. These cases require wrappers to the API described previously.

In most cases, an additional function call or the introduction of a macro suffices. These convert the original function call into a series of steps. First, the new code creates a new custodian and sets the appropriate limit upon it. Then the code invokes the original call in a new subthread and waits for the computed value. In short, a function call is converted into a short-lived subprocess with appropriate constraints.

A disadvantage of this approach involves the creation of the temporary custodian and thread, which involves an obvious performance penalty. However, it seems unlikely that converted calls will appear in tight loops. (Calls in tight loops typically execute quickly, and thus are unlikely to have behavior requiring memory constraints.)

Even if speed is not an issue, functions requiring thread identity will not work using this method. For example, security mechanisms might grant privileges only to one particular thread, but not subthreads of that thread. By running the function in a new thread, we lose any important information stored implicitly in the original thread. While cases such as this seem rare, we are working on a solution for this problem.

6.2.2 Producer-Based Accounting

The only situation in which our system does not clearly subsume existing technologies is when producer-based ac-

counting is required. In other words, process A provides data to process B, and charges for the data should always go to A, regardless of the relationship between the two.

We have not encountered in practice an example where producer-based accounting makes sense, and we conjecture that they are rare. Even so, this problem might often reduce to a rate-limiting problem rather than a memory usage problem. In other words, B does not so much wish to constrain the memory use of A as much as it wants to constrain the amount of data that it receives at one time. Suitable rate-limiting protocols should suffice for this case. Further, using a weak reference allows a program to hold data without being charged for it. Since weak references do not cause the data contained in them to be retained by the collector, the accounting system does not charge their holders with their data.

Another possible scenario involves the use of untrusted libraries for creating and maintaining data structures. We can imagine using an off-the-shelf library to hold complex program data, rather than writing the data structures from scratch. In using such a library, we can imagine wanting to ensure that its structures do not grow unreasonably. Again, this situation seems unlikely. Programs typically do not trust important data to untrusted data structure libraries. Further, it is unclear how the program would continue to run after a constraint violation kills its own data structures.

7. IMPLEMENTATION

Implementing consumer-based accounting requires only straightforward modifications to a typical garbage collector. Our approach requires the same mark functions, root sets and traversal procedures that exist in the collector. The primary change is in the organization and ordering of roots. A second involves the method of marking roots, and the final involves a slight change in the mark procedure.

First and foremost, accounting requires a particular organization of roots. Before accounting, the roots must be ordered so that root A appears before root B if the custodian responsible for A descends from the custodian responsible for B. Second, we mark and fully propagate each individual root before moving on to the next one. Figure 3 outlines these steps.

By placing this partial order on the roots and forcing full propagation before moving on to the next root, we provide the full guarantee described previously. If object x is reachable by roots from custodians A and B, where B is a descendent of A, then the mark propagation will select B's root first due to the partial order. When the root or roots associated with A come around, x has already been marked and no further accounting information is gathered for it.

Minor modifications to the mark procedure alleviate one further potential problem. These stop mark propagation when threads, custodians and weak boxes are reached, in order to match the behavior outlined in section 4.

Doing this process alongside the collector allows one pass to perform both accounting and collection, and works in many cases. Our first implementation of memory accounting used this methodology. However, in cases where the set of collectable objects include threads, creating appropriate orderings becomes difficult. Since the accounting mechanisms requires the marking of threads and the collector does not know which threads are live, determining how to proceed becomes tricky.

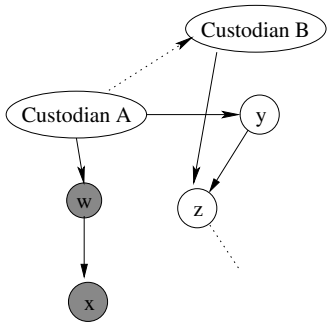

Figure 4: A potential heap layout, mid-collection. The grayed objects have been marked by the collector.

In such cases, a second pass implementing memory accounting suggests itself. This second pass occurs after garbage collection but before control returns to the mutator. Our implementation of such a system suggests that the additional effort required above that of the all-in-one solution remained minimal. Again, using much of the existing collector scaffolding saves considerable amounts of work and time. This two-pass style may increase slowdowns noticeably in some cases, but we have not noticed any significant difference between the two implementations in practice.

7.1 Incremental Collection

Our implementation of memory accounting builds on an existing precise, stop-the-world garbage collector for MzScheme. We believe that the basic algorithm described above transfers easily to other languages and other virtual machines running stop-the-world collectors. Obviously, languages without a custodian mechanism must transfer their granularity to some other entity, but the core of the algorithm should remain the same.

To see the problem in combining incremental and accounting collectors, consider the example in Figure 4. In this case, we have two custodians (A and B) and four objects (w, x, y and z), and two of the objects – w and x – have been marked by the collector. If the collector now turns control over to the mutator, the mutator may then modify the heap so that the link from y to z is destroyed and a new link from x to z is created. At this point, an unmodified incremental collector will account z to B, which violates the guarantee in section 4.

In incremental collectors, the collector uses either read barriers or write barriers to prevent the mutator from putting references from unscanned (*white*) objects to scanned (*gray* or *black*) objects [1, 15]. Using a write-barrier technique, the collector must either *gray* the referring object (scheduling it for repropagation once the collector resumes) or gray the referred-to object (forcing it into scanned space).

In both cases, any references by the newly-grayed object will be placed in the collector's mark queue. In order to support accounting, we require two modifications. First, the trap routine must place these objects at the head of the mark queue, causing them to be marked immediately once the collector resumes. Second, the trap routine must annotate these objects with the custodian to which the referring object was accounted. By combining these, the collector

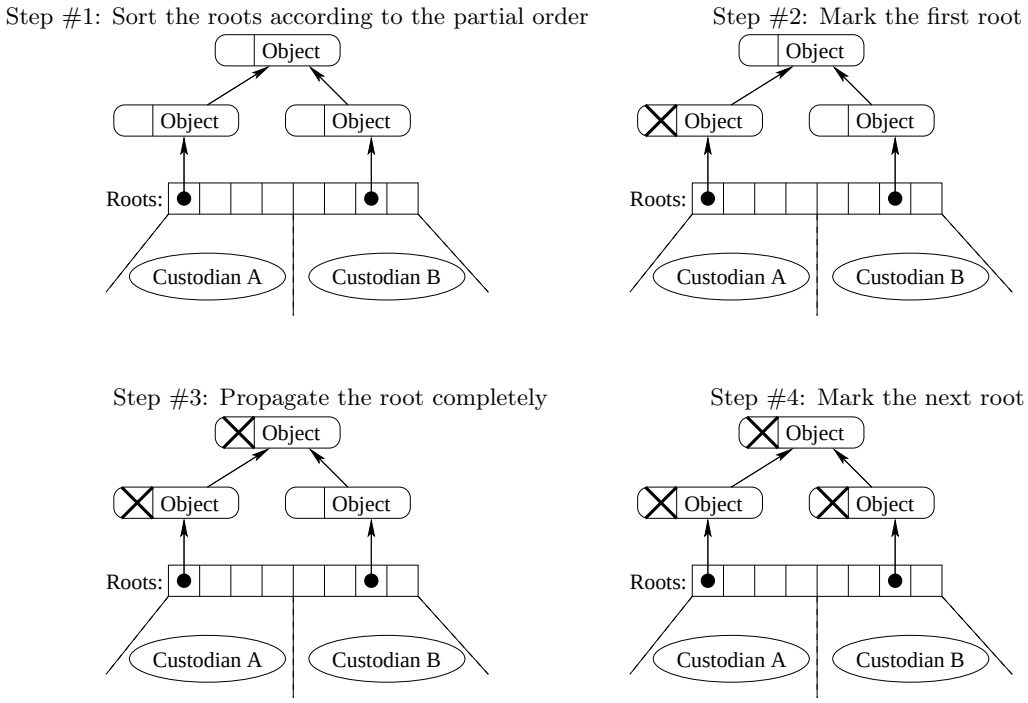

Figure 3: The four steps of the accounting procedure.

accounts these objects to the appropriate custodian immediately after it resumes execution.

Since these objects are put at the head of the queue, there is no chance of a parent reaching these objects before its child, thus insuring the guarantee described in section 4. Since the modification simply modifies the order of objects added to the queue in the trap function, a modified version of an incremental algorithm will halt if the unmodified version halts.

7.2 Benchmarks

Our example applications are not particularly amenable to benchmarking, but quick tests show either no slowdown outside of measurement noise or slowdown of under 10%. In this section, we use a set of microbenchmarks instead. While microbenchmarks do not generalize particularly well, we feel that they are appropriate in this case. Since we are testing for slowdown, rather than speedup, the inordinate stress these benchmarks place on the memory system supply us with worst case behavior. Most actual programs should experience penalties no worse than those reported.

We implemented our accounting system atop a garbage collector for MzScheme. By default, MzScheme uses a conservative collector, due to historical needs. We plan on migrating the system to using precise collection by default in the coming year or two, and our accounting system is built upon the newer precise collection infrastructure. In order to more completely convey the penalties involved from accounting, we include three sets of benchmarks.

The first set of benchmarks, in figure 5, outline the base penalty for using precise collection instead of the Boehm conservative collector [6]. We compare the original MzScheme C source with the Boehm collector to a system using C source modified to enable precise collection which is also linked to the Boehm collector. The penalty for the extra work (largely involving keeping track of data values on the stack) is nontrivial but acceptable for most common tasks.

We then compare the modified source with the Boehm collector to the base, nonaccounting precise collector. Results are given in figure 7. These results show precise collection to be strictly faster than the conservative collector (linked against the same modified source). Thus we claim that the cost of accounting, reported below, is not small simply because the collector is abnormally slow.

Finaly, we added memory accounting to the base precise collector to determine the penalty for accounting. We provide two accounting collectors for comparative purposes. The first generates accounting information only on major collections, while the second gathers accounting information on every collection. We provide only the second in our distribution system, but the first is an intuitive optimization we include for completeness. The results are given in figure 7.

A brief description of each benchmark follows:

- *earley*: Earley's context-free parsing algorithm. Given a simple ambiguous grammar, it generates all the parse trees for a short input.
- *gcbench*: A synthetic benchmark originally written in Java by John Ellis, Pete Kovac and Hans Boehm.
- *graphs*: Performs an enumeration of directed graphs, making heavy use of higher-order procedures.
- *lattice*: Enumeration of lattices of monotone maps between lattices.

Name	Boehm	Mod. Boehm	Slowdown
earley	189 (<1%)	258 (1.94%)	36.51%
gcbench	13560 (<1%)	20350 (<1%)	50.07%
graphs	240007 (<1%)	280330 (<1%)	16.8%
lattice	79368 (<1%)	123970 (<1%)	56.2%
nboyer	66627 (<1%)	111535 (<1%)	67.4%
nucleic	2585 (<1%)	3995 (<1%)	54.55%
perm	1685 (1.54%)	3197 (<1%)	89.73%
sumperm	2021 (<1%)	3082 (<1%)	52.5%
mergesort	29211 (<1%)	52431 (<1%)	79.49%
sboyer	66434 (<1%)	111451 (<1%)	67.76%

Figure 5: Comparison of the standard Boehm conservative collector with the Boehm collector linked to our precise-collection modified C source code. The modifications required to get standard C to work with a precise collector which may move objects around slows down the total efficiency of the system noticeably. All times given in milliseconds, with standard deviations as a percentage of the total given in parenthesis.

Name	Mod. Boehm	Precise	Speedup
earley	258 (1.94%)	202 (2.48%)	21.71%
gcbench	20350 (<1%)	18405 (1.24%)	9.56%
graphs	280330 (<1%)	230559 (<1%)	17.75%
lattice	123970 (<1%)	118392 (<1%)	4.5%
nboyer	111535 (<1%)	94175 (<1%)	15.56%
nucleic	3995 (<1%)	3936 (<1%)	1.48%
perm	3197 (<1%)	2788 (1.33%)	12.79%
sumperm	3082 (<1%)	2530 (<1%)	17.91%
mergesort	52431 (<1%)	42521 (<1%)	18.9%
sboyer	111451 (<1%)	95338 (<1%)	14.46%

Figure 6: Comparison of the standard Boehm conservative collector linked to the precise-collection modified C source with the base precise collector. Precise collection outperforms the Boehm collector in all cases. All times given in milliseconds, with standard deviations as a percentage of the total given in parenthesis.

- *nboyer*: Bob Boyer's theorem proving benchmark, with a scaling parameter suggested by Boyer, some bug fixes noted by Henry Baker and others, and rewritten to use a more reasonable representation for the database (with constant-time lookups) instead of property lists (which gave linear-time lookups for the most widely distributed form of the boyer benchmark in Scheme).
- *nucleic*: Marc Feeley et al's Pseudoknot benchmark.
- *perm*: An algorithm for generating lists of permutations, involving very large amounts of allocation.
- *sumperm*: Computes the sum of a set of permutations over all permutations.
- *mergesort*: Destructively sorts a set of permutations.
- *sboyer*: The nboyer benchmark with a small modification to use shared conses.

All benchmark data was gathered over 50 trials of each benchmark program for each collector, with the means and standard deviations given. The test system is a Pentium IV running FreeBSD 4.6 and PLT Scheme 205.5.

We find that accounting – even accounting on every collection – costs little compared to the runtime of each program. Additional overheads in these benchmarks show no more than a 13% slowdown, which matches our day-to-day experience. Most programs show between 4 and 8%, with a few programs experiencing little to no slowdown at all.

In order to simplify this presentation, we have included standard GC performance benchmarks. Unfortunately, these are not concurrent. However, we find that accounting performance on concurrent programs with multiple custodians behaves in the same way as single process programs. Since the accounting system only marks each object in the heap once, regardless of the number of active custodians, this behavior is not surprising.

The difference between accounting only on major collections and accounting on every collection seems minor in most cases. However, accounting on only major collections presents important disadvantages. Gathering information much less frequently creates long delays in the detection of memory constraint violations. In some cases, a sufficient enough delay occurs to cause unfortunate side effects, such as thrashing when the underlying virtual memory system is forced to go to disk for more memory. We feel that more prompt detection of these constraints far outweighs the minor performance degradation.

On the other hand, we have had no problems gathering accounting information on every collection. Typically, minor collections happen frequently enough that any delay in detecting memory violations is trivial. The only cases where collections do not occur frequently are when threads do not allocate much memory at all.

8. RELATED WORK

We previously reported preliminary results for our accounting system [14]. Since the preliminary report, we have fully implemented the accounting system, reconsidered some subtle design points, and performed a thorough evaluation of our system.

Other recent research focuses on providing hard resource boundaries between applications in order to prevent denial of service attacks. The KaffeOS [2] for Java provides the ability to precisely account for memory consumption by applications. MVM [7], Alta [3], and J-SEAL2 [5] all provide similar solutions, as do JSR-121 [13] and .NET application domains [11], but in all cases these boundaries constrain interprocess communication. In highly cooperative applications, or in situations requiring some amount of dynamic flexibility in sharing patterns, these systems may present significant barriers to simple development.

Generally, the existing work on resource controls – including JREs [8] and research on accounting principals in operating systems address only resource application, which does not adequately provide the full range of tools we believe modern programmers require.

Our notion of custodians arose in parallel with work on resource containers [4] for general-purpose operating systems. The accounting system in some ways adds memory back as a resource. However, the resource container system again accounts for memory based on the producer, rather than the consumer, of the data. Similarly, the resource container system requires explicit management of the containers, thus

Name	Base collector	Major-collection	Slowdown	Every collection	Slowdown
earley	202 (2.48%)	193 (2.59%)	-4.46%	215 (2.33%)	6.44%
gcbench	18405 (1.24%)	17988 (<1%)	-2.27%	19626 (<1%)	6.63%
graphs	230559 (<1%)	232068 (<1%)	0.65%	259050 (<1%)	12.36%
lattice	118392 (<1%)	118483 (<1%)	0.08%	121901 (<1%)	2.96%
nboyer	94175 (<1%)	95906 (<1%)	1.84%	98956 (<1%)	5.08%
nucleic	3936 (<1%)	3979 (<1%)	1.09%	4418 (<1%)	12.25%
perm	2788 (1.33%)	2689 (<1%)	-3.55%	2910 (<1%)	4.38%
sumperm	2530 (<1%)	2465 (<1%)	-2.57%	2596 (<1%)	2.61%
mergesort	42521 (<1%)	40522 (<1%)	-4.7%	42917 (<1%)	0.93%
sboyer	95338 (<1%)	97336 (<1%)	2.1%	99497 (<1%)	4.36%

Figure 7: Comparison of the accounting collectors compared to the base collector. Standard deviations as a percentage of the total time are given in parenthesis. Times given in milliseconds.

requiring explicit management of heap partitions and thus memory management. These do provide some measure of precision, but restrict the free flow of data unacceptably.

Price et al. [12] present the only other consumer-based, partition-free accounting system, which was developed in parallel with our system. They present no results for practical applications. Moreover, their policy for shared objects is to rotate the mark order for roots, which would not provide a sufficient guarantee for any of our motivating applications. Price et al. also present a concept of "unaccountable references." These references effectively block the marking phase of the accounting mechanism. The rationale for these objects is to block a malicious process A from passing an inordinately large object to process B in an attempt to get it killed. It is unclear, however, what advantages unaccountable references have over normal weak references.

9. CONCLUSION

Current accounting mechanism require programmers to partition their heaps, which creates communications problems, protocol problems, and manual memory management problems. We have shown that this additional work is unnecessary in a garbage collected environment. Our partition-free, producer-based accounting system provides the constraints programmers need without interfering with the way they design or write their programs.

We have described three example applications. Implementing memory constraints for each of these, given only existing technologies, would have required careful protocol design, complicated marshaling code, or explicit memory management. Using our accounting system, each required very little effort and only minimal changes.

More generally, our analysis suggests that our accounting system works for many accounting paradigms. In many cases, our system should work better, since it enables more direct communication among tasks.

Finally, our system exists and is available with PLT Scheme 205.5 and higher. It is fast enough to be used in production quality systems, typically causing only a 4 to 8% slowdown,

Acknowledgments

The authors would like to gratefully acknowledge Wilson Hsieh for discussions leading to this work.

10. REFERENCES

[1] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 11–20. ACM Press, 1988.

[2] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000. USENIX.

[3] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. In *Proceedings of the USENIX 2000 Technical Conference*, pages 197–210, San Diego, CA, June 2000.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. ACM Symposium on Operating System Design and Implementation*, Feb. 1999.

[5] W. Binder, J. G. Hulaas, and A. Villazón. Portable resource control in java: The J-SEAL2 approach. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 139–155, 2001.

[6] H.-J. Boehm. Space efficient conservative garbage collection. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[7] G. Czajkowski and L. Daynès. Multitasking without compromise: a virtual machine evolution. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 125–138, 2001.

[8] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, 1998.

[9] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. `http://download.plt-scheme.org/doc/`.

[10] M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (or revenge of the son of the lisp machine). In *International Conference on Functional Programming*,

pages 138–147, 1999.

[11] E. Meijer and J. Gough. Technical overview of the common language runtime.

[12] D. W. Price, A. Rudys, and D. S. Wallach. Garbage collector memory accounting in language-based systems. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2003.

[13] Soper, P., specification lead. JSR 121: Application isolation API specification, 2003. `http://www.jcp.org/`.

[14] A. Wick, M. Flatt, and W. Hsieh. Reachability-based memory accounting. In *2002 Scheme Workshop*, Pittsburgh, Pennsylvania, October 2002.

[15] T. Yuasa. Realtime garbage collection on general-purpose machines. *Journal Of Systems And Software*, 11:181–198, 1990.