Reachability-Based Memory Accounting

Adam Wick
awick@cs.utah.edu

Matthew Flatt
mflatt@cs.utah.edu

Wilson Hsieh
wilson@cs.utah.edu

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

ABSTRACT

Many language implementations provide a mechanism to express concurrent processes, but few provide support for terminating a process based on its resource consumption. Those implementations that do support termination generally charge the cost of a resource to the principal that *allocates* the resource, rather than the principal that *retains* the resource. The difference matters if principals represent distinct but cooperating processes.

In this paper, we present preliminary results for a version of MzScheme that supports termination conditions for resource-abusing processes. Unlike the usual approach to resource accounting, our approach assigns fine-grained (per-object) allocation charges to the process that retains a resource, instead of the process that allocates the resource.

1. MOTIVATION

Users of modern computing environments expect applications to cooperate in sophisticated ways. For example, users expect web browsers to launch external media players to view certain forms of data, and users expect a word processor to support active spreadsheets embedded in other documents. In a conventional operating system, however, programmers must exert considerable effort to integrate applications. Indeed, few software developers have the resources to integrate applications together as well as, for example, Adobe Acrobat in Microsoft's Internet Explorer.

Implementing cooperating applications in a conventional OS is difficult because the OS isolates applications to contain malfunctions. Cooperating applications must overcome this built-in isolation. In contrast, language run-time systems (a.k.a. "virtual machines") typically rely on language safety, rather than isolation, to contain malfunctions. Since VMs otherwise play the same role as OSes, and since they lack a bias towards isolation, safe VMs seem ideally suited as the platform for a next generation of application software.

Mere safety, however, does not provide the level of protection between applications that conventional OSes provide. Although language-based safety can prevent a program from trampling on another program's data structures, it cannot prevent a program from starving another process or from leaking resources. Regardless of the degree of cooperation, a practical OS/VM must track each application's resource consumption and prevent over-consuming applications from taking down the entire system.

A variation on conventional isolation can certainly enable resource tracking in a VM. For example, the VM can restrict values passed from one process to another to those values allocated within a certain pool of memory [1]. This compromise provides something better than a traditional OS, in that a suitably allocated value can be passed directly and safely between applications. Nevertheless, this kind of isolation continues to interfere with cooperation: even if a program can move values from one allocation pool to another, explicit accounting with allocation pools amounts to manual memory management as in `malloc` and `free`. This manual management encourages narrow communication channels; in order to foster communication, applications must be free to exchange arbitrary data with potentially complex allocation patterns.

We are investigating memory-management techniques that place the responsibility for accounting with the run-time system, instead of the programmer, while still enabling control over an application's memory use. The essential idea is that a garbage collector can account for memory use using reachability from an application's roots. Thus, an application is charged not for what it allocates, but for what it retains. This differentiation is critical in systems where one application may use memory allocated by another application. The central design problem is how to deal with these shared values usefully and efficiently.

We present preliminary results on our exploration, based on a new garbage collector for MzScheme [7]. Our results suggest that a garbage collector can maintain usefully precise accounting information with a low overhead, but that the implementation of the rest of the VM requires extra care to trigger reliable termination of over-consuming processes. This extra care is of the same flavor as avoiding references in the VM that needlessly preserve values from collection.

Section 2 describes the existing notion of "process" within MzScheme, and Section 3 describes our new API for resource enforcement. Section 4 describes in more detail possible accounting policies behind the API, including the two that we have implemented. Section 5 reports on our implementations, and Section 6 reports on our experience with them. Section 7 presents performance results.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

2. PROCESSES IN MZSCHEME

In MzScheme, no single language construct encompasses all aspects of a conventional process. Instead, various orthogonal constructs implement different aspects of processes:

- *Threads* implement the execution aspect of a process. The MzScheme thread function takes a thunk and creates a new thread to execute the thunk.

  The following example runs two concurrent loops, one that prints "1"s and another that prints "2"s:

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

- *Parameters* implement process-specific settings, such as the current working directory. Each parameter is represented by a procedure, such as current-directory, that gets and sets the parameter value. Every thread has its own value for each parameter, so that setting a parameter value affects the value only in the current thread. Newly created threads inherit initial parameter values based on the current values in the creating thread.

  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))
```

  Meanwhile, the current-directory setting of other executing threads is unaffected by the above code.

- *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 children 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. A thread can, however, change its current custodian at any time. In the above example, since *child-custodian* is current when the child thread is created, the child is placed into the management of *child-custodian*. Thus, (custodian-shutdown-all *child-custodian*) reliably terminates the child thread. In addition, if *child-custodian* is the only custodian accessible in *child-work-thunk*, then any custodian, thread, port, or network listener created by the child is reliably shut down by (custodian-shutdown-all *child-custodian*).

  Evaluating (current-custodian) immediately in *child-work-thunk* would produce *child-custodian*, because the initial parameter values for the child thread are inherited at the point of thread creation. The child thread may then change its current custodian at any time, perhaps creating a new custodian for a grandchild thread. Again, if *child-custodian* is the only custodian accessible in *child-work-thunk*, then newly created custodians necessarily fall under the management of *child-custodian*.

MzScheme includes additional constructs to handle other process aspects, such as code namespaces and event queues, but those constructs are irrelevant to accounting.

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

3. ACCOUNTING API

Accounting information in MzScheme depends only on custodians and threads. Accounting depends on custodians because they act as a kind of process ID for termination purposes. In particular, since the motivation for accounting is to terminate over-consuming processes, MzScheme charges memory consumption at the granularity of custodians. Accounting also depends on threads, because threads encompass the execution aspect of a process, and the execution context defines the set of reachable values. Thus, the memory consumption of a custodian is defined in terms of the values reachable from the custodian's threads.

We defer discussion of specific accounting policies until the next section. For now, given that accounting is attached to custodians, we define a resource-limiting API that is similar to Unix process limits:

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

  Typically, *cust1* and *cust2* are the same custodian, but distinguishing the accounting center from the cost center 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 there comes a time when *cust1* would be unable to allocate *need-k* bytes, 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.

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*.

4. ACCOUNTING POLICIES

4.1 Reachability

As described in the previous section, we define a custodian's memory consumption in terms of the values reachable from the custodian's threads, as opposed to the values originally allocated by the threads. In addition, we require that the custodian hierarchy propagates accounting charges: if a custodian B is charged for a value, then its parent custodian is charged for the value as well.

Generally, reachability for accounting coincides with reachability for garbage collection. In particular, a value is not charged to a custodian if it is accessible through only weak pointers. Finalization poses no problem for accounting, because every finalizer in Mzscheme is created with respect to a *will executor*. Running a finalizer requires an explicit action on the executor, which means that a finalized object can be charged to the holder of the finalizer's executor.

Accounting reachability deviates from garbage-collection reachability in one respect. If a value is reachable from thread A only because thread A holds a reference to thread B, then B's custodian is charged and not A's (unless A's custodian is an ancestor of B's). Similarly, if a value is reachable by A only through a custodian C, then C is charged instead of A's custodian.

This deviation makes intuitive sense, because holding a reference to another process does not provide any access to the process's values. Moreover, this deviation is necessary for making accounting useful in our test programs, as we explain in Section 6.

4.2 Sharing

In a running system, some values may be reachable from multiple custodians. Different accounting policies might allocate charges for shared values in different ways, depending, on the amount of sharing among custodians, the hierarchical relationship of the custodians, the original allocator for a particular value or other factors. Among the policies that seem useful, we have implemented two:

- The *precise* policy charges every custodian for each value that it reaches. If two custodians share a value, they are both charged the full cost of the value. For example, in figure 1, objects w and z will be charged to both custodians A and B, object x will be charged to both custodians B and C, and object Y will be charged only to custodian C.
- The *blame-the-child* policy charges every value to at least one custodian, but not every custodian that reaches the value. The main guarantee for blame-the-child applies to custodians A and B when A is an ancestor of B; in that case, if A and B both reach some value, then A is charged if and only if B is charged. Meanwhile, if B and C share a value but neither custodian is an ancestor of the other, then at most one of them will be charged for the object. For example, in figure 1, object Y will be charged only to custodian C as in the precise policy. Also, since custodian B is a child of custodian A, B will necessarily be charged for W and Z. In the case of X, since there is no ancestral relationship between B and C, no guarantees are given as to which will be charged.

The precise policy is the most obvious one, and seems easiest to reason about. We have explored the blame-the-child policy, in addition, because it can be implemented more efficiently than the precise policy (at least in theory).

The blame-the-child policy, despite its imprecision, can work with custodian-limit-memory to control the memory consumption of a single sand-boxed application. Since the sand-boxed application will share only with its parent, accounting will reliably track consumption in the sand box.

Blame-the-child is less useful with custodian-limit-memory in a setting of multiple cooperating children. In that case, a well-behaved, cooperating application might incur all of the cost of all shared values, triggering the termination of the over-charged child (possibly leaving the rest stuck, lost without a collaborator). However, blame-the-child always works well with custodian-require-memory. With memory requirements instead of memory limits, how memory charges are allocated among children does not matter.

One policy that we have not explored is a variant of precise that splits charges among sharing custodians. For example, suppose that x custodians share a value of size y. With splitting, each of the x custodians would be charged y/x. This policy is normally considered troublesome, because terminating one of the x custodians triggers a sudden jump in the cost of the other $x - 1$. Like blame-the-child, though, this policy might be useful with custodian-require-memory. We have not explored the cost-splitting policy because it seems expensive to implement, and it does not appear to offer any advantage over blame-the-child.

4.3 Timing

Ideally, a policy should guarantee the termination of a custodian immediately after it violates a limit or requirement. A naive implementation of this guarantee obviously cannot work, as it amounts to a full collection for every allocation.

The policies that we have implemented enforce limits and requirements only after a full collection. Consequently, a custodian can overrun its limit temporarily. This temporary overrun seems to cause no problems in practice, because a custodian that allocates lots of memory (and thus might violate limits or requirements) tends to trigger frequent collections. Furthermore, a failure in allocation for any custodian triggers a garbage collection which will then terminate usage offenders to satisfy the allocation.

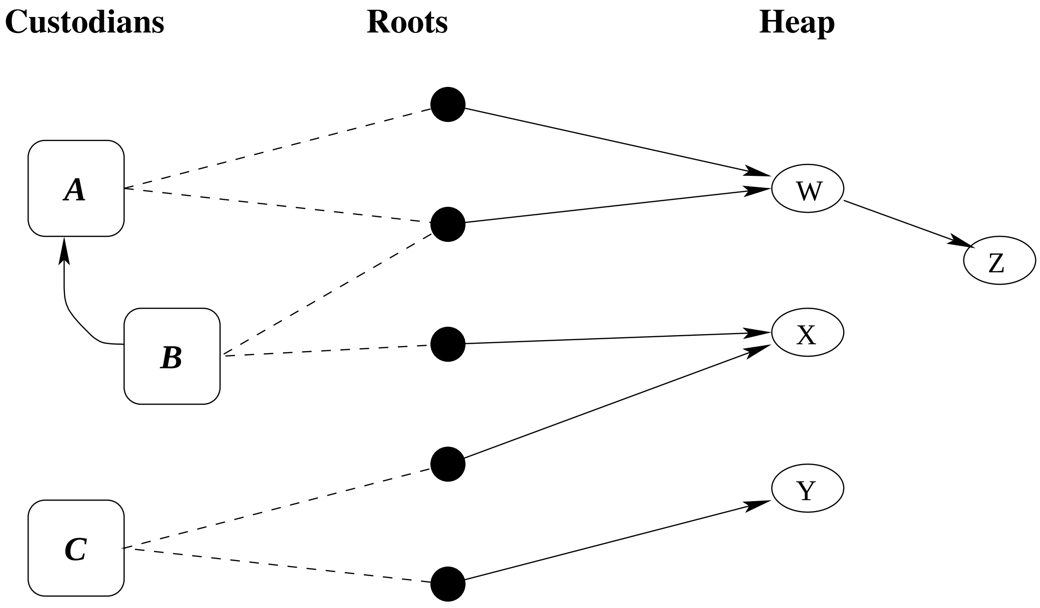

Figure 1: An example set of custodians and roots with a small heap

One potential problem is that a child overrun could push its parent past a limit, where terminating the child earlier might have saved the parent. Another problem is that a child overrun may occur at a time when custodians cannot be safely terminated. These potential problems have not appeared in practice, primarily because programmers cannot know the exact cost of values and must include significant safety margins. Nevertheless, the problems merit further investigation.

5. IMPLEMENTATION

The implementation of both the precise and blame-the-child policies proceeds roughly as follows:[2]

1. When a thread is created, the creating thread's current custodian is recorded in the new thread.

2. The collector's mark procedure treats thread objects as roots and as it marks from each thread, it charges the thread's custodian.

3. After collection, the collector checks the accumulated charges against registered memory limits and requirements. The collector schedules custodians for destruction (on the next thread-scheduling boundary) according to the comparison results.

Our two implementations differ only in the details of step 2. We first describe the implementation of precise accounting, then the implementation of blame-the-child accounting. Finally, we discuss the impact of generational garbage collection on the algorithms.

5.1 Precise Accounting

For precise accounting, the collector reserves space in the header of each object to record the object's set of charged custodians (CS_o in figure 2). During collection, the mark queue contains objects paired with the custodian set to be charged for the object. Initially, the charged set for all objects is the empty set. The initial mark queue contains all thread objects, where each thread is paired to the charged set containing only the thread's custodian.

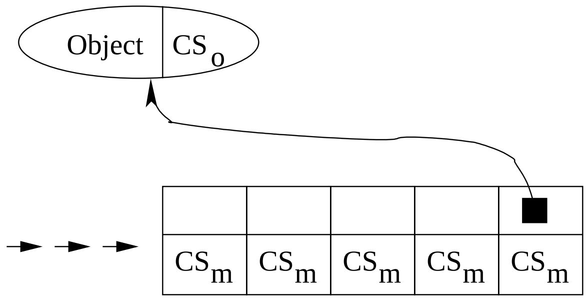

Figure 2: Mark queue with an object

When mark propagation reaches an object (see figure 2), the charged set in the object's header (CS_o) is compared to the charge used in marking (CS_m). If the charge set for marking is a subset of the charged set CS_o in the object header, no further work is performed for the object.[3] Otherwise, the union of the sets is computed and installed into the object's header, and charges for the object are shifted from the old set (if it is non-empty) to the unioned set. Marking continues with the object's content using the unioned set. After marking is complete, all garbage objects have an empty charged set, and the charges accumulated for each set can be relayed back to the set members.

[2]The algorithms described should work in any collection system. We use the terminology of a mark/sweep style collector to simplify the description.

[3]If the object contains a charge set, then it has been marked, and the mark propagation has either already been done or is queued. Since the item's charged set is a superset of the mark's charge set, then no additional information is available and no further work needs to be done.

In the case of a single custodian, the above algorithm degenerates to plain garbage collection, since the only possible charge sets are the empty set and the set containing the one custodian. In the case of c custodians, collection potentially requires c revisions to the charged set of every object. Thus, in the worst case, collection requires $O(c * r)$ time, where r is the size of reachable memory and c is the size of the set of all custodians. An entire heap containing only a single linked list with every thread pointing to the head of the list is an example of this worst case.

5.2 Blame-the-child

Unlike precise accounting, blame-the-child accounting requires only linear time in the amount of live memory. Roughly, the blame-the-child implementation works in the same way as the precise implementation, except that objects with non-empty charge sets are never re-marked. This change is enough to achieve linear time collection.

To completely implement the blame-the-child policy, the collector sorts the set of custodians before collection so that descendents precede ancestors. Then, the threads of each custodians are taken individually. Each thread is marked and the marks are propagated as far as possible before continuing with the next threads. Due to this ordering, objects reachable from both a parent and child will be first reached by tracing from the child's threads, and thus charged to the child. Once collection is complete, charges to child custodians are propagated back to their parents.

In our implementation, the blame-the-child implementation also incurs a smaller per-object overhead, since object headers need not contain a charge set. During marking, exactly one custodian is charged at a time, so that charges can be accumulated directly to the custodian. Each object needs only a mark bit, as in a normal collector.

A naive implementation of blame-the-child allows an obvious security hole. By creating sacrificial children, a malevolent custodian may arbitrarily delay its destruction when it uses too much memory. Such children would have pointers back into the malevolent custodian's space so that they would be blamed for its bad behavior. These, then, would be killed instead of the parent.

Several possible mechanisms can be used to keep this from happening, and we simply chose the easiest one from an implementation perspective. They are:

1. Place an order on the list of limits and requirements so that older custodians are killed first. In this case, the parent will be killed before the children, so creating sacrificial children is useless.

2. Kill every custodian that breaks a limit or requirement, rather than just one. Since a child's usage is included in the parent's usage, both will be killed.

3. Choose a random ordering. In this way, a malevolent program would have no guarantee that the above tactic would work.

Our implementation chooses the second tactic.

5.3 Generational Collection

After a full collection is finished and accounting is complete, comparing charges to registered limits and requirements is simple. Therefore, the collector can guarantee that a custodian is terminated after the first garbage collection cycle after which a limit or requirement is violated. This implies that there may be some delay between the detection of a violation and the actual violation. However, if the program is allocating this delay will be small, as frequent allocation will quickly trigger a garbage collection.

Accounting information after a minor collection is necessarily imprecise, however, since the minor collection does not examine the entire heap. Previously computed sets of custodians for older objects might be used regardless of changes since their promotion to an older generation. This old information may arbitrarily skew accounting. Worse, in the blame-the-child implementation described above, the collector does not preserve charges in object headers, so there is no information for older generations available to partial collections (except those that reclaim only the nursery).

Our implementation therefore enforces limits and requirements only after a full collection. This choice can delay enforcement by several collections, but should not introduce any new inherent potential for limit overruns, since overruns must lead to a full collection eventually.

6. EXPERIENCE

To determine the usefulness of our accounting policies in realistic environments, we wrote and modified several programs to take advantage of accounting. One program simply tests the ability of a parent to kill an easily sand-boxed child. A second program, DrScheme, tests child control where the parent and child work closely together. A third program, a web server allowing arbitrary servlet plug-ins, tests child control with some cooperation among the children.

6.1 Simple Kill Test

In the simple kill test, the main process creates a single sub-custodian, places a 64 MB limit on the sub-custodian's memory use, and creates a single thread in the sub-custodian that allocates an unbounded amount of memory:

```
(let ([child-custodian (make-custodian)])
  (custodian-limit-memory child-custodian
    (* 64 1024 1024) child-custodian)
  (current-custodian child-custodian)
  (thread-wait ; blocks until the thread completes
    (thread (lambda ()
              (let loop ()
                (+ 1 (loop)))))))
```

Since accounting works, the child custodian is destroyed, which in turn halts the child thread, and the entire program completes. If accounting were not successful, then the program would not terminate. Under both of our accounting system implementations, we find this program terminates. Unfortunately, it terminates several garbage collection cycles after the limit is actually violated.

Although simple, this program presents two items of interest. First, it shows that the blame-the-child policy can work, and that it allows the natural creation of parent/child pairs where the parent wishes to limit its children. Second, the program shows that generational collection does delay the detection of resource overruns.

Safety nets in our garbage collector assure that a program does not run out of available memory before its limit is noticed, but in systems with tight memory requirements, our technique may not be acceptable. We are investigating ways to detect overruns more quickly.

6.2 DrScheme

The DrScheme programming environment consists of one or more windows, where each window is split into two parts. The top part of the window is used to edit programs. The bottom part is an interactive Scheme interpreter loop where the program can be tested. Each interpreter (one per window frame) is run under its own custodian. With a single line of code, we modified DrScheme to constrain each interpreter to 16 MB of memory.

Initial experiments with the single-line change did not produce the desired result, even with precise accounting. After opening several windows, and after making one interpreter allocate an unbounded amount of memory, *every* interpreter custodian in DrScheme terminated. Investigation revealed the problem:

- Each interpreter holds a reference into the DrScheme GUI. For example, the value of the parameter current-output-port is a port that writes to the text widget for the interaction half of the window. The text widget holds a reference to the whole window, and all open Drscheme windows are chained together.
- Each window maintains a reference to the interpreter thread, custodian, and other interpreter-specific values, including the interpreter's top-level environment.

Due to these references, every interpreter thread reaches every other interpreter's data through opaque closures and objects, even though programs running in different interpreters cannot interfere with each other. Hence, in the precise accounting system, every thread was charged for every value in the system, which obviously defeats the purpose of accounting.

Correcting the problem required only a slight modification to DrScheme. We modified it so that a window retains only weak links to interpreter-specific values. In other words, we disallow direct references from the parent to the child. Thus a child may trace references back to the parent's values, but will never trace these references back down to another child.

Finding the parent-to-child references in DrScheme—a fairly large and complex system—required only a couple of hours with garbage-collector instrumentation. The actual changes required only a half hour. In all, five references were changed: two were converted into weak links, two were extraneous and simply removed, and one was removed by pushing the value into a parameter within the child's thread.

Breaking links from parent to child may seem backward, but breaking links in the other direction would have required far too much work to be practical. For example, we could not easily modify the interpreter-owned port to weakly reference the DrScheme window. The port requires access to many internal structures within the GUI widget. Indeed, such a conversion would amount to the file-descriptor/handle approach of conventional operating systems—precisely the kind of design that we are trying to escape when implementing cooperation.

6.3 Web Server

In the DrScheme architecture, children never cooperate and share data. In the web server, however, considerable sharing exists between child processes. Whenever a server connection is established, the server creates a fresh custodian to take charge of the connection. If the connection requires the invocation of a servlet, then another fresh custodian is created for the servlet's execution. However, the servlet custodian is created with the same parent as the connection custodian, not as a child of the connection custodian, because a servlet session may span connections. Thus, a connection custodian and a servlet custodian are siblings, and they share data because both work to satisfy the same request.

The precise accounting system performs well when a servlet allocates an unbounded amount of memory. The offending servlet is killed right after allocating too much memory, and the web server continues normally.

The blame-the-child system performs less well, in that the servlet kill is sometimes delayed, but works acceptably well for our purposes. The delayed kill with blame-the-child arises from the sibling relationship between the connection custodian and the servlet custodian. When the servlet runs, the connection is sometimes blamed for the servlet's memory use. In practice, this happens often. The result is that the connection is killed, and then the still-live memory is not charged to the servlet until the next garbage collection. This example points again to the need for better guarantees in terms of the time at which accounting charges trigger termination, which is one subject of our ongoing work.

7. PERFORMANCE EVALUATION

Memory accounting incurs some cost, with trade-offs in terms of speed, space usage, and accounting accuracy. To measure these costs, we have implemented these two memory accounting systems within MzScheme.[4] Our collector is a generational, copying collector[8] implemented in C. This collector is designed for production-level systems; it can handle all situations that the default MzScheme garbage collector handles, including finalizers which may resurrect dying objects. For analysis purposes, the collector can be tuned statically to behave as one the following:

- **NoAccnt**: The base-line collector. No memory accounting functionality is included in this collector.
- **Precise**: The base-line collector plus the memory accounting system described in section 5.1.
- **BTC**: The base-line collector plus the memory blame-the-child accounting system described in section 5.2.

We evaluate the space usage, accuracy and time penalty of the **BTC** and **Precise** collectors on the following benchmark programs:

- **Prod**: An implementation of a producer/consumer system, with five producers and five consumers paired off. A different custodian is used for each producing or consuming thread. This case covers situations wherein sibling custodians share a large piece of common data; in this case, they share a common queue.
- **Kill**: A basic kill test for accounting. A child custodian is created and a limit is placed on its memory use.

[4] Accounting builds on the "precisely" collected variant of MzScheme, instead of the "conservatively" collected variant.

Test	Precise # of owner sets	Precise Additional required space	BTC # of owner sets	BTC Additional required space
Web	360	60,054 bytes	360	30,570 bytes
Prod	35	3,842 bytes	21	1,130 bytes
DrScheme	15	6100 bytes	9	5076 bytes
PSearch	4	266 bytes	3	186 bytes
Kill	2	146 bytes	2	146 bytes

Figure 3: Additional space requirements for accounting.

	NoAccnt Time	NoAccnt S.D.	BTC Time	BTC S.D.	BTC % slowdown	Precise Time	Precise S.D.	Precise % slowdown
Web	1.30	0.05	1.77	0.06	36.2%	1.80	0.06	38.5%
Prod	2.60	0.05	1.31	0.04	n/a	1.41	0.04	n/a
DrScheme	23.10	0.14	23.55	0.11	1.7%	43.19	1.73	87.0%
PSearch	2.33	0.12	2.41	0.12	3.4%	2.42	0.13	3.9%
Kill	n/a	n/a	1.74	0.03	n/a	1.76	0.04	n/a

Figure 4: Timing results in seconds of user time with standard deviations. Where applicable, the table provides a percentage slowdown relative to the *NoAccnt* collector. All benchmark programs were run on a 1.8Ghz Pentium IV with 256MB of memory, running under FreeBSD 4.3 and MzScheme (or DrScheme) version 200pre19.

Under the child custodian, memory is then allocated until the limit is reached. This case covers the situation wherein proper accounting is necessary for the proper functioning of a program.

- **PSearch**: A search program that seeks its target using both breadth-first and depth-first search and uses whichever it finds first. This case is included to consider situations where there are a small number of custodians, but those custodians have large, unshared memory use.
- **Web**: A web server using custodians. This test was included as a realistic example where custodians may be necessary. The server is initialized, and then three threads each request a page 200 times. Every thread on the server side which answers a query is run in its own custodian.
- **DrScheme**: A program, run inside DrScheme, that creates three custodian/thread pairs and starts a new DrScheme process in each.

7.1 Space Usage

Regardless of the implemented policy, some additional space is required for memory accounting. Space is required internally to track the custodian of registered roots, and to track owner sets. In the case of **Precise**, additional space may be required for objects whose headers do not contain sufficient unused space to hold the owner set information for the object.

In our tests, the space requirements usually depend on the number of owner sets. Figure 3 shows the amount of space required for each of our test cases. These numbers show the additional space overhead tracking, roughly, the number of owner sets in the system. The numbers for **DrScheme** do not scale with the others because the start-up process for the underlying GUI system installs a large number of roots.

As expected, the additional space needed for precise accounting is somewhat larger than the space required for blame-the-child accounting. This space is used for union sets (owner sets which are derived as the union of two owner sets), and the blame-the-child implementation never performs a set union. The difference thus depends entirely upon the number of custodians and the sharing involved.

The MzScheme distribution includes a garbage collector that is tuned for space. In particular, it shrinks the headers of one common type of object, but this shrinking leaves no room for owner set information. Compared to the space-tuned collector, the **NoAccnt** and the accounting collectors require between 15% and 35% more memory overall.

7.2 Accuracy

To check the accuracy of memory accounting for different collectors, we tested each program under the precise system and compared the results to the blame-the-child system. The results were exactly as expected: the blame-the-child algorithm accounts all the shared memory to one random child. For example, in **DrScheme**, precise accounting showed that around 49 MB of data was shared among the children. Under **BTC**, one of the custodians (and not necessarily the same one every time) and its parent were charged 49 MB, but the other two child custodians were charged only for local data (around 80 KB each).

7.3 Time efficiency

To measure the trade-off between the accuracy of accounting information and the execution speed of the collector (and hence the program as a whole), we recorded the total running time of the test programs. Figure 4 shows the results of these benchmarks.

In every case, precise accounting takes additional time. The amount of additional time depends on the number of custodians, the amount of sharing among the custodians, and the size of the data set. In **Web**, **Prod**, **PSearch**, and **Kill**, the custodians and heap are arranged so that the additional penalty of precise accounting (that is, the penalty beyond that of **BTC** accounting) is minimal. The greatest slowdown in those cases, around two percent, is for **Web**. In contrast, for cases where there is considerable sharing and the heap is large, the penalty for precise accounting can be quite large. **DrScheme** fits this profile, and the slowdown for precise accounting is predictably quite high.

Blame-the-child accounting also incurs a performance penalty. In both **DrScheme** and **PSearch**, the penalty is small. In **Web**, the penalty is significant. The difference between the former two tests and the latter one is primarily in the number of owner sets they use. The penalty difference, then, may result from cache effects during accounting. Since owner-set space usage is kept in a table, this table may become large enough that it no longer fits in cache. By reading and writing to this table on every mark, a large number of owner sets imply considerably more cache pressure and hence cache misses. In ongoing work, we are investigating this possibility.

The strange case in our results is **Prod**. In this case, the work of accounting actually speeds up the program. In ongoing work we are trying to determine the cause of the speed-up.

8. RELATED WORK

Recent research has focused on providing hard resource boundaries between applications to prevent denial-of-service. For example, the KaffeOS virtual machine [1] for Java provides the ability to precisely account for memory consumption by applications. Similar systems include MVM [5], Alta [2], and J-SEAL2 [4]. This line of work is limited in that it constrains sharing between applications to provide tight resource controls. Such restrictions are necessary to execute untrusted code safely, but they are not flexible enough to support high levels of cooperation between applications.

More generally, the existing work on resource controls—including JRes [6] and research on accounting principals in operating systems, such as the work on resource containers [3]—addresses only resource allocation, and does not track actual resource usage.

9. CONCLUSIONS

We have presented preliminary results on our memory-accounting garbage collection system for MzScheme. Our approach charges for resource consumption based on the retention of values, as opposed to allocation, and it requires no explicit declaration of sharing by the programmer. Our policy definitions apply to any runtime system that includes a notion of accounting principles that is tied to threads,

In the long run, we expect our blame-the-child accounting policy to become the default accounting mechanism in MzScheme. It provides accounting information that seems precise enough for many applications, and it can be implemented with a minimal overhead.

The main question for ongoing work concerns the timing of accounting checks. Our current implementation checks for limit violations only during full collections, and the charges for a terminated custodian are not transferred until the following full collection. Both of these effects delay the enforcement of resource limits in a way that is difficult for programmers to reason about, and we expect that much better guarantees can be provided to programmers.

A second question concerns the suitability of weak links for breaking accounted sharing between a parent and child, and perhaps between peers. The current approach of weakening the parent-to-child links worked well for our test programs, but we need more experience with cooperating applications.

The collectors described in this paper are distributed with versions 200 and above of the PLT distribution of Scheme for Unix.[5] Interactive performance of the accounting collectors is comparable to the performance of the default collector, although some pause times (particularly when doing precise accounting) are noticeably longer.

10. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[5] Configure with `--enable-account` and make the `3m` target.