Operating System Support for General-Purpose Memory-Adaptive Applications

Neal H. Walfield
Johns Hopkins University
neal@cs.jhu.edu

ABSTRACT

Many programs could improve their performance by adapting their memory use according to availability. If memory is available, a web browser or DNS server could use a larger cache; if there is memory pressure, they could use a smaller one. Memory adaptations are also becoming increasingly important for scalability: server consolidation is being done more aggressively and end-users want to run their application on a wider-range of hardware configurations. Today, memory-based adaptations are fragile: general-purpose operating systems do not indicate how much memory a process should or could use.

Enabling efficient adaptations requires rethinking how memory is allocated among competing programs, and adding a feedback mechanism that allows applications to make informed adaptations. In this paper, we present the design and implementation of a minimum-funding revocation scheduler for memory. We describe a novel algorithm to compute the amount of memory available to each resource principal based on its scheduling parameters and the current configuration, explain how to communicate this information to the principals, and show how they can exploit it. We also present a new mechanism to account shared memory based on access frequency. We demonstrate the effectiveness of the techniques by showing that multiple applications changed to exploit this information use the full memory available to them, and smoothly vary their demand as availability changes. This also results in significant increases in throughput relative to the conservative management techniques currently used.

Categories and Subject Descriptors

D.4.10.a [**Operating Systems**]: Support for Adaptation; D.4.2 [**Operating Systems**]: Storage Management—*Main Memory*

Keywords

memory management, adaptation, feedback, accounting

1. INTRODUCTION

An adaptation's effectiveness is bounded by the quality of the environmental feedback that informs it. Commodity operating systems make little such information available. The practical result is that programmers either avoid memory adaptations or they make their adaptations conservative, e.g., aggressively shrinking caches if objects are unused for a short time [26]. The underlying problem is that excessive memory use leads to thrashing, which usually yields a net loss in performance.

Historically, most programs' demands have been a function of their input. For instance, the resource requirements of a text-book implementation of quick sort are primarily determined by its input data. For such programs, a memory-management strategy based on the working-set model of program behavior is sufficient to ensure good throughput and reasonable demand on system resources, and will avoid thrashing when possible.

Two trends are making memory adaptations more important. First, server consolidation is becoming increasingly common. A program that allocates a static amount of memory at start up decreases the system's ability to dynamically reallocate memory and limits that program's ability to respond to spikes in demand. Second, the range of hardware configurations, which commodity applications are expected to run on, is growing: users want their regular desktop applications to run on their smart phones [14], for instance.

To better support these scenarios, we propose that operating systems provide feedback to applications. Feedback will allow applications to better gauge how much memory they should use, and to proactively adapt to changes in availability. This type of information is particularly useful for programs that use memory caches for computed data, and garbage collectors, whose collection overhead is often a function of the number of live objects, not heap size [3].

We consider the use of a minimum-funding revocation scheduler to manage memory. We create an allocation hierarchy and have processes indicate the relative importance of each of their children. When there is memory pressure, the scheduler starts from the root and selects the child computation that most exceeds its fair share. This is applied recursively. The selected principal is then signalled that it must free

some memory or it will soon be paged.

Using the same scheduling parameters, we present an algorithm that computes how much memory is available to each principal. This is not simply how much memory it is currently using plus the amount of unused memory in the system: in the steady state, we expect most memory to be allocated; moreover, this devolves to a first-come first-served policy. Instead, we include how much memory a process could claim from others as well as allocation trends.

Both of the previous algorithms rely on accurate accounting. Many systems do not track the principals to which memory is allocated. This is not trivial to add: a mechanism is needed to determine how to account and revoke shared memory. We propose accounting shared memory according to access frequency. To accomplish this, we rotate the ownership of a shared frame among its users according to their respective access frequency. To reduce crosstalk, when a frame is selected for revocation, it is added to the end of the eviction list. If another principal accesses it prior to its eviction, it is saved and accounted to the new principal.

Results are promising. Using our algorithm, we show that multiple aggressively-adaptive memory applications are able to fully utilize memory according to their relative priorities and are able to adapt as new principals enter and leave the system.

Contributions In summary, this paper makes three contributions: a design for a minimum-funding revocation memory scheduler; an algorithm to compute how much memory is available to each process; and, a mechanism to account shared memory based on access frequency. Our benchmarks show that the algorithms are effective for multiple aggressively-adaptive memory applications.

2. BACKGROUND

We consider several application scenarios where feedback is useful. We then briefly look at hardware trends and their potential impact on application programming.

2.1 Application Loads

Many programs and libraries could exploit a feedback mechanism to improve performance. We survey the DEENS DNS server, image caching, database caching, and garbage collectors.

In [24], Madhavapeddy et al. demonstrate that DEENS, a DNS server written in OCaml, can respond to 10% more queries per second than BIND, the industry standard DNS server. They also show that adding a query cache to DEENS, which requires just four lines of code, results in more than a 50% increase in throughput on their benchmarks. They experimented with two policies: a cache that grows without bound, i.e., entries are never evicted, and, a cache that is flushed whenever a garbage collection is initiated. Using these techniques, the benchmark's throughput went from approximately 14,000 queries per second to over 25,000 queries per second and over 22,000 queries per second, respectively. Both of these policies have weaknesses: whereas allowing the cache to grow without bound is susceptible to thrashing, dropping the cache at the start of every collection results in unnecessarily aggressive cleaning. A better management strategy is one that sizes the cache according to the amount of available memory.

Source Image		Decompress		Decompress and Scale	
Dimensions	Size	Time	Size	Time	Size
1600×1200	429K	0.13s	5.5M	0.40s	2.5M
3008×2000	2.4M	0.43s	17.2M	0.95s	2.2M

Table 1: The time to decompress and the time to decompress and scale images to fit in a 764×706 area. Performed on an Intel Core2 Quad core running at 2.4GHz.

Decompressing and scaling images is expensive and can be usefully cached. Table 1 shows the time it takes to decompress and to decompress and scale a 2 megapixel and a 6 megapixel image on a modern desktop machine. According to Nielsen [25], 0.1 seconds is the limit after which a user no longer feels that the system reacts instantly. In all cases, the time to compute the data to display is longer. To improve the user experience, if there is a chance that an image may be viewed again and there are idle resources, caching the data would seem like a good strategy. Again, the difficulty is in sizing the cache: a too large cache may result in unacceptable paging. Programs such as GQview and gThumb cache the last two viewed images and prefetch one image. Oftentimes, more memory could be used. For mobile internet devices, however, even this modest cache size will often be too large. Ideally, the cache should grow to fill the available memory and the application should be able to shrink it when there is pressure.

Databases use caches to improve performance. In addition to overriding the operating system's buffer management policy to take advantage of application-specific knowledge [29], they also cache computed data. For instance, query plans and query results (both intermediate and final) are non-trivial to compute and can often be effectively reused [9]. Part of the management of such caches should be sizing them according to memory availability.

Many garbage collectors could also use feedback to size the heap, which can improve performance. The sweep phase of a garbage collector starts at the roots of the process (the statically-known live data) and traces all pointers to identify the set of live objects. Any other object must be dead since it is unreachable: no live object references it. When using a compacting collector, only the live objects are touched during the sweep phase. The result is that for many types of garbage collectors, the time to perform a collection is proportional to the number of live objects. Since the number of live objects is independent of the heap size, using a larger heap results in less frequent collections and an overall decrease in the time spent collecting [3]. Yang et al. observe that this holds in practice [32].

A feedback mechanism could also replace other *ad hoc* management strategies such as using time outs to control thread pool management, e.g., in Apache, and how long to keep rendered objects such as widgets around when there are no references [26]. Likewise, it could be used to manage code

caches for dynamically generated code [18] and control when to reap free objects in a user-level slab allocator [7]. Although the tear-down and recreation time may be small, if there is no pressure, these operations simply represent unnecessary work.

2.2 Range of Execution Environments

Whereas in the past the execution environment was often predictable, today, that is changing due to server consolidation and increasing popularity of low-powered consumer devices running general-purpose applications with dynamic workloads.

Server consolidation represents a problem as many programs are configured to use a fixed amount of memory. In a consolidated environment, this effectively partitions significant portions of memory. This reduces the opportunity for better handling bursts in demand: if many unrelated application instances are hosted on a single machine and one experiences a spike, it could exploit idle resources to improve its response if they were available. It also decreases the potential degree of consolidation: due to access patterns, many servers see long periods of inactivity followed by spikes in demand [10].

In the consumer sector, netbooks, mobile internet devices (MIDs) and smart phones running general-purpose operating systems have become popular. In 2008, in reaction to consumer demand for the ability to run their usual programs on their iPhones, Citrix announced a virtualization environment to run Windows applications on the iPhone [14]. Even while consumers embrace these low-end devices, high-end desktops continue to offer more resources for a potentially-improved experience. For applications and the system to be able to provide maximum quality of service, applications need to take advantage of and not exceed available resources. Feedback regarding memory availability is a step in this direction.

3. DESIGN

We now present an approach to scheduling memory that is appropriate for memory-adaptive applications, and show how to reclaim memory when there is pressure. We then turn to computing availability given the available scheduling parameters. Finally, we discuss how to account memory and provide a solution for dealing with shared memory.

3.1 Allocation

Computing the amount of memory available to an application is not as simple as adding the application's current allocation and the amount of unallocated memory in the system. In their stable state, most systems have little unallocated memory: if memory is not allocated to an application, it is allocated to the buffer cache. This results in a first-come first-served allocation policy: when the first adaptive application sees that there is memory available, it allocates it; subsequent adaptive applications will see that there is no memory available and use a minimal footprint.

The goal of an adaptation is to change an application's behavior such that it makes the most of the memory available to it. The effect is to maximize the application's expected utility. To maximize expected utility among competing entities, memory must be distributed according to their respective expected utility. The difficulty is in determining each application's expected utility function. Ultimately, these functions depend on a human's real-world goals, which are potentially changing, and are difficult for an external observer such as a central memory manager to infer.

One approach to maximizing expected utility system wide is to allocate memory hierarchically and to allow principals to control how much each of their children may use. In such a system, the system administrator distributes memory among users, users distribute memory among the programs they start, and the programs distribute memory among any sub-computations, etc. This maximizes expected utility as the parent is usually a computation's primary stakeholder.

If programs actually allocate physical memory to each child, this complicates dynamic reallocation of memory and potentially makes the revocation protocol difficult. First, programs must actually understand the memory needs of each child and the trade-offs associated with different allocations. This is often difficult for the program itself to know and reason about, never mind the parent. Second, applications must also manage deallocations. This means that when a principal must yield memory, it must determine from which child memory should be reclaimed. As programs may not be trusted, this quickly becomes complicated.

The revocation issue can be solved by instead of having parents directly control resource allocations, they provide an expected-utility function to the memory manager for each child. Using expected-utility functions, parents have the same control and similar expressiveness, however, the manager is able to act without having to consult them for every allocation and deallocation request. This increases the agility of reallocation: the memory manager is able to directly revoke memory from one leaf on one side of the tree and reallocate it to another leaf on the other side; the shift need not ripple through the system as it does when ancestors impose on every allocation and deallocation. This approach has the added advantage that the expected-utility functions are in terms of allocation preferences, which are independent of the actually available resources.

Expected-utility functions still require that parents understand how each child computation uses resources: these functions are articulated in terms of low-level resources, which are difficult for humans—programmers as well as users—to reason about. Relative preferences approximate expected-utility functions (which, in the end, describe preference relations) but are independent of the actual resources. The idea is that the resource manager provide more resources to computations that are more important. Parents then use the high-level feedback that they do understand to tweak the preferences as required.

Waldspurger proposes the use of a minimum-funding revocation scheduler to manage memory in his work on proportional-share scheduling [30]. The idea is that when memory needs to be revoked, the resource principal that provides the least amount of funding per unit is chosen to yield memory. The underlying principle is essentially a reformulation of eco-

nomic supply and demand: as demand increases, prices increase and those who cannot afford the new prices must do without the goods. (As described in the related work section, Waldspurger only evaluates this work using a simulation.)

3.1.1 Scheduling Parameters

To use a minimum-funding revocation scheduler to implement a proportional-share scheduler using the process hierarchy, resource principals assign each of their children a weight. This weight parameter determines the amount of its parent's memory to which each child is entitled.

It is useful to be able to express strict priorities. For instance, some computations should always be able to preempt others. This is the case for the user's input handler and window manager, which the user relies on to be able to control misbehaving programs. Also, some programs should really only be run if there is nothing else to do, for instance, a file indexer.

Priorities can be expressed using weights: high-priority tasks can be given weights that are orders of magnitude larger than normal-priority tasks, which can in turn be given weights that are orders of magnitude larger than background tasks. Alternatively, priorities can be made first class. Conceptually, a priority level can be thought of as shorthand for an infinite multiplier.

Although a priority parameter does not add anything to the expressiveness of the scheduling parameters, it is convenient and makes working with weights easier. If weights are the same size as the machine word, there is a greater chance of overflowing when using them in computations.

In addition to these parameters, demand should also be considered when allocating memory. As a goal is to maximize memory utilization, allocating memory to a principal that has not asked for it is a waste if some other principal could use it. Thus, demand should be used to inform the *maximum* allocation for a principal.

An application's demand should also be used in determining its *minimum* allocation: if a process is allocated less memory than that required to hold its working set, it will begin to thrash [11].

We identify three approaches to deal with this potential thrashing. First, if all related resources are carefully accounted and scheduled, thrashing can simply be ignored with minimal quality-of-service crosstalk: in this case, a thrashing program only hurts itself. Second, the process can be suspended until there is sufficient memory for it to run. Whether this is better than thrashing is usually best decided by the parent, which can either be notified by way of a signal, or can specify the desired policy as an explicit scheduling parameter. Finally, an application's working set can be weighted differently from the rest of its memory. This can also be specified by the parent. For instance, two weight parameters could be provided, one for weighting the memory in the working set, and a second for inactive (and presumably, opportunistic) memory.

Algorithm 1 Selecting a principal to revoke resources from.

```
1: function SelectVictim(principal)
2:     ws_factor ← 1
3:     loop
4:         for P ⊂ { children(principal) grouped by
               priority, lowest to highest } do
5:             weight_per_frame ← ∞
6:             for p ∈ P do
7:                 if p.alloced > p.ws/ws_factor then
8:                     t ← p.weight / (p.alloced − p.ws/ws_factor)
9:                     if t < weight_per_frame then
10:                        weight_per_frame ← t
11:                        victim ← p
12:            if victim ≠ nil then
13:                return SelectVictim(victim)
14:        ws_factor ← ws_factor · 2
```

These approaches can also be combined. For instance, a parent might indicate that a child's working set should only count half and if that is still not enough to ensure that it continues to run without significant paging, it should be suspended.

3.1.2 Allocation Contract

There are two general ways to allocate resources. Either a principal can request an allocation (or a list of useful allocations and preferences [19]) before starting a computation, e.g., 100 MB for 20 seconds, and the resource manager can admit it or not, or a principal can request resources lazily, i.e., on demand.

Obtaining an allocation before committing to a computation ensures that the resources required for the computation are available for the duration of the computation. A difficulty with this is that programmers must be able to estimate the resources required for the computation *a priori*. However, if a programmer is successful, this can improve quality-of-service as the operating system will not preempt the resources while they are being used, which can result in paging or potentially complicated application recovery mechanisms, e.g., rolling-back state and using an approach that requires less memory.

When resources are allocated on demand, a program allocates resources at the point in time that it needs them. This is how resources are usually scheduled on Unix-like systems. This approach makes bringing up applications easier. Typically, these allocations are not accompanied by a temporal guarantee; the system can reclaim them at any time, which improves systems scheduling flexibility but complicates application efforts to ensure a particular quality of service.

Due to the difficulty that we perceive with programmers specifying memory allocations and the fact that most applications are written to use an on-demand interface, we concentrate on the latter and leave integrating memory reservation as future work.

3.1.3 Revocation Algorithm

Algorithm 1 presents a memory revocation approach for a system using hierarchical allocations, which have no mini-

mum duration (i.e., memory can be revoked at any time), and in which each principal is assigned a weight and a priority. The algorithm considers a principal's working set by initially completely excluding it. If this would result in not being able to revoke any memory, it gives the working set less weight and repeats.

The algorithm is called by the memory manager, which passes it the root resource principal. It starts by setting the working set factor to 1. Given the set of principals in the lowest priority class, it selects the principal such that the weight per frame accounted to it minus its working-set size corrected by the working set factor is smallest. If this is not positive for all principals in the priority class, this is repeated using the next-highest-priority class. If this is the case for all priority classes, the algorithm doubles the working set factor (giving less weight to each principal's working set) and repeats. Otherwise, it has found a victim. If that principal has no children, the algorithm terminates. Otherwise, it recurses.

Given a victim, a local revocation scheme can be used. For instance, some least-recently-used frames accounted to the principal may be revoked.

3.2 Availability

The allocation hierarchy can be used to compute the amount of memory that is approximately available to a principal. Publishing this information allows an adaptive program to reliably determine if it can grow its allocation without being paged and when it should shrink its allocation to avoid being paged.

As a first approximation, a principal's availability is its share of the resources. Consider, however, a high-priority principal and a low-priority one. Using this measure, the former's share is all of the memory and the latter's share is none. Although the high-priority principal could preempt resources allocated to the lower-priority principal, it need not. If it appears not to want to do so, reporting to the low-priority principal that no memory is available to it is counter-productive.

A more useful approximation considers not only the principals' scheduling parameters but also their current allocations. Assuming that most programs are generally quiescent, a reasonable approximation is to define a principal's availability as the amount of memory that the principal could allocate. That is, a principal's availability is what it has allocated, what is unallocated and what it could preempt. This is easy to calculate when just considering the weight and priority scheduling parameters. A principal can preempt nothing from higher-priority principals and everything from lower-priority principals. From principals with the same weight, it can preempt memory from those principals with a smaller weight-per-frame than its weight-per-frame. In particular, it can preempt memory from them until their respective weight-per-frame equals its own. Algorithm 2 shows how to compute the total amount of memory available to each principal using this idea.

This strategy effectively captures the amount of memory a process could use, however, it does not enable proactive adaptation to memory pressure. Consider what happens as some principal begins to allocate memory: once all memory has been allocated, it will begin to preempt memory from others. A principal's reported availability will only reflect the change after the system has already preempted its memory. The availability metric can be improved by also incorporating allocation trends.

Algorithm 2 Compute the amount of available memory based on the amount a principal could preempt.

```
1: procedure AVAILABLE(principal)
2:     avail ← principal.avail
3:     for P ⊂ children(principal) grouped and sorted
           by priority, highest to lowest do
4:         for p ∈ P do
5:             avail -= p.alloc
6:         for p ∈ P do
7:             for q ∈ P do
8:                 if q.weight/q.alloc ≤ p.weight/p.alloc then
9:                     w += q.weight
10:                    a += q.alloc
11:            p.avail ← p.weight·a/w + avail
12:    for p ∈ children(principal) do
13:        Available(p)
```

Assuming that recent allocations are generally a sign of future allocations, i.e., that a principal will try to allocate (or deallocate) an amount of memory in the near future proportional to the amount of memory it recently allocated, we can first simulate the allocations arriving at an effective allocation and then compute each principal's availability using Algorithm 2 but using each principal's effective allocation in place of its actual allocation.

Algorithm 3 shows how to compute the effective allocation of each principal. The basic idea is that it determines how much memory each principal would have available if each principal allocated the same number of frames as it recently allocated. The presented algorithm allocates just one frame at a time. This is inefficient but can be optimized to run in $O(|principals^2|)$ time by moving frames in batches.

3.2.1 Publishing Availability

Availability can be exposed either via a polling or a subscription interface. The former is useful for applications that have occasional adaptation points. If an application can almost always adapt, e.g., a garbage collector can collect at almost any time, and a cache can always shrink, the subscription model enables agile adaptations with low overhead.

Using a subscription model, it is also possible to send messages when memory is about to be revoked from a principal. If the system holds some memory is reserve, it is possible to continue to satisfy allocations while giving the victim principal some time to free memory. This reduces the need somewhat for Algorithm 3.

3.3 Accounting

The ability to accurately implement the previous algorithms relies on accurate accounting information: if the system does

Algorithm 3 Compute each principal's effective allocation assuming it tries to allocate as much as it recently allocated.

```
 1: procedure EffectiveAllocation(principal)
 2:     avail ← principal.avail
 3:     for p ∈ children(principal) do
 4:         if p.rec_alloc < 0 then                          ▷ Freeing
 5:             p.eff_alloc ← p.alloc + p.rec_alloc
 6:             avail += p.rec_alloc
 7:         else
 8:             p.eff_alloc ← p.alloc
 9:     for P ⊂ children(principal) grouped and sorted
            by priority, highest to lowest do
10:         for p ∈ P do
11:             avail -= p.alloc
12:         for p ∈ P do
13:             alloc ← p.rec_alloc
14:             while alloc > 0 do
15:                 if avail > 0 then
16:                     avail − −
17:                     p.eff_alloc + +
18:                 else
                        Select q ∈ (children(principal)
                        s.t. eff_alloc > 0,
19:                     priority is minimal, and
                        weight/eff_alloc is minimal)
20:                     if p = q then
21:                         break
22:                     q.eff_alloc − −
23:                     p.rec_alloc − −
24:                 alloc − −
25:     for p ∈ children(principal) do
26:         EffectiveAllocation(p)
```

not track which frames are accounted to a principal, determining its weight per frame is guesswork; further, if it is not known which frames a principal is using, it is impossible to implement a local revocation policy.

One difficulty with memory accounting comes from shared memory. A possible solution is to have all users of memory split the cost evenly. There are two issues with this approach. First, it is unclear what should be done when a principal is selected to yield memory and a shared frame is chosen for eviction. Second, accounting a frame to all users requires storage proportional to the number of users.

We propose that a frame be assigned to a single principal at a time and that ownership be rotated according to access frequency. The result is that the amount that any one principal pays is proportional to its use. Also, the resource manager is able to use a fixed size data structure for managing a frame.

Detecting use can be accomplished by occasionally unmapping shared frames and then assigning the frame to a new owner on the next access. A possible disadvantage of this approach is that it increases the number of soft page faults.

Timing when shared frames are unmapped is important. Periodically unmapping shared frames could allow malicious programs to free ride by timing their access near the end of an epic, that is, after the frame has likely been claimed. To mitigate such an attack, the time between unmaps should be determined stochastically.

4. IMPLEMENTATION

To evaluate the proposed algorithms, we implemented a prototype system called Viengoos. Viengoos is an activation-based [2, 27], object-capability microkernel [12, 28].

4.1 Primordial Objects

The Viengoos virtual machine augments the hardware interface with seven objects: folios, data pages, capability pages, threads, activities, messengers and end points. Objects are designated using capabilities, which include a weak bit (thus distinguishing two access rights, a "strong" access right and a "weak" access right).

Folios A folio holds the meta-data for 128 objects. Metadata was separated from objects to ensure that an object's size remains an easy-to-manage power-of-two. The metadata is explicitly represented to ensure that it is properly accounted and to reduce sharing.

Pages A page contains data, uninterpreted by the kernel.

Cappages A cappage contains 256 capability slots. A capability slot may contain a capability. Cappages are also used in the construction of address spaces. In this role, they function as page tables.

Threads A thread encapsulates an execution context. It includes capability slots that designate the address-space root, and current activity. It also includes space for a copy of the platform's register file. Multiple threads can execute in the same address space by using the same address-space root.

Activities An activity encapsulates a resource principal and a scheduling policy. Like a resource container [5], it is orthogonal to an execution context. All allocations are done with respect to some activity. An activity's parent is the activity to which it is accounted. An activity includes the weight and priority scheduling parameters. These may only be changed by way of a strong capability.

Because an activity that has children may also allocate memory, a mechanism is required to determine if the memory should be revoked from the parent rather than one of its children. This is treated in a uniform manner by also having so-called child-relative priority and weight scheduling parameters. These can be set via a weak capability. This does not enable any interference as the child-relative parameters do not affect how much memory the activity is entitled to but how to distribute the memory available to the activity between an activity and its children.

When an activity is destroyed, all resources allocated to it are revoked. As any child activities are allocated out of an activity's own resources, this also destroys any children. By running each program as a separate activity, destroying an activity is a convenient way to ensure that when it is done, all resources it allocated and did not explicitly arrange to

save with some other entity—including its child programs—are freed.

Messengers A messenger encapsulates a message and includes capabilities, untyped data, and a capability slot indicating the thread to optionally activate on message receipt and delivery. Messengers enable reliable, asynchronous IPC.

End Points An end point indirects access to messengers. This enables multi-threaded servers to expose a single entry point. Our prototype does not include an end-point implementation.

4.2 Resource Allocation

Whereas storage (e.g., data pages) is allocated and deallocated explicitly, memory frames are allocated implicitly, on demand. When an object that is not in memory is accessed, Viengoos transparently brings it into memory and accounts the memory to the appropriate activity. In the case of a fault, this is the activity designated by the capability in the thread's activity slot; for RPCs, the activity is provided explicitly.

It is important to emphasize that the activity to which memory is accounted is not necessarily the same as that to which the corresponding storage is accounted. This matches common usage. The storage used to hold the system libraries and programs is usually accounted to the system administrator but primarily used by program instances running on behalf of users. Similarly, the storage for a user's files is accounted to a principal representing the user, however, the principals using the data in memory are program instances running on behalf of the user.

4.3 Accounting

A frame is the basic unit of memory allocation. A frame caches an object on backing store. Associated with each frame is a management data structure. At system start, the number of frames is calculated and the meta-data data structures are allocated. This is possible as the amount of precious frame meta-data (the data that cannot simply be discarded without ill effect, unlike, e.g., the software TLB contents) is known *a priori* and not dependent on the number of users.

When a frame is allocated to an activity, we say that the activity is the *claimant* for the memory, or that it has *claimed* it. An activity also claims memory that it causes to be allocated, e.g., when it accesses an object that is on backing store. An activity claims memory if it accesses memory that is inactive. The justification for this is that some activity must pay for the memory until it is freed either by freeing the associated storage, or paging it out. If the memory becomes inactive, then this is an indication that the activity has not used it in a while.

To deal with shared memory, maintain accurate accounting information, and only require a small, fixed amount of meta-data, we account a frame to a single activity at a time but allow it to migrate among users so as to distribute the cost according to access frequency. To do this, when a frame is accessed by an activity other than its claimant, the frame is marked *shared*. Occasionally, shared frames are unmapped and marked *floating* (and, the shared mark is removed). The next activity to access such a frame claims it and removes its floating status. In this way, the cost of shared memory is divided according to access frequency.

Currently, shared frames are marked as floating every two seconds. This value was chosen arbitrarily. A higher frequency ensures a more accurate distribution of the costs but also results in an increased overhead due to the added soft faults.

Viengoos uses a simple page ager to track which frames are active and which are inactive. The ager runs at 2 Hz. Every two seconds, it also unmaps shared pages, recalculates the amount of memory available to each active activity, and gathers statistics.

4.4 Scheduling

Each activity has priority and weight variables that the parent can set as well as child-relative priority and weight variables that it can set. An activity's need and its working set also determine the amount of resources it has allocated.

We approximate an activity's working set by classifying pages that have been referenced in the last two seconds (the active pages) as being in some program's working set. This is, of course, a particularly poor metric as the working set is determined in a program's virtual time [11]. This will affect, for instance, interactive programs, which, when blocked waiting for input, will appear to have an empty working set. More investigation is required to determine whether this is sufficient or if a better approximation should be used.

4.5 Eviction Policy

Viengoos maintains four memory pools: in-use, free, dirty, and available. The in-use pool consists of memory that is claimed; the free pool of memory that is not allocated; the dirty pool of memory that has been selected for replacement but that needs to be flushed to disk; and, the available pool of memory that has been selected for replacement and can be reused immediately.

If, when allocating memory, the amount of in-use memory exceeds 7/8s of the total memory, the pager is activated. The pager migrates enough frames from the in-use pool to the other pools such that the in-use pool does not exceed 13/16s of the total memory.

Dirty and available frames remain accounted to the last claimant so that further costs can be correctly accounted. Such frames, however, are not counted towards the activity's claimed frames. When a dirty or available frame is accessed, it is claimed and added back to the in-use pool. This is similar to VMS's second-chance strategy [23].

The frames to evict are selected using a two-stage algorithm. First, a victim activity is selected according to algorithm 1. Then, some frames are selected for eviction from that victim using a least-recently-used policy. This process is repeated until enough frames have been removed from the in-use pool such that the in-use pool does not exceed 13/16s of the total memory.

4.5.1 Evicting Frames

Given a victim, only the frames claimed directly by it are considered for eviction. The number of frames that it must yield is a function of the degree of its excess relative to its siblings and the number of active frames. Specifically, this is the activity's frames minus the priority group's frames multiplied by the activity's weight divided by the priority group's weight. This is capped by the number of inactive frames, which avoids thrashing.

Note that all objects are considered for replacement, not just data pages. The only objects that are treated specially are activities: an activity object is only evicted if it has no claims and no child activity is in memory.

5. EVALUATION

We evaluate our framework using two main benchmarks. First, we modified the Boehm garbage collector to only perform collections when the amount of allocated memory approaches or exceeds the amount of available memory. Second, we developed a cache manager, which can either use a fixed-sized cache or can adapt the cache to the amount of available memory. Both resulted in significant performance improvements. We also conducted an experiment to determine the cost of the proposed shared-memory accounting scheme. The results show that the overhead is minimal.

Our test system had an Intel Core2 Quad core running at 2.4GHz with a 4MB L2 cache and 2GB of RAM. All benchmarks were run in uniprocessor mode. For the tests on GNU/Linux, we used Debian 5.0.

5.1 Garbage Collection

For our experiments, we used version 7.0 of the Boehm collector [6]. By default, the Boehm collector schedules a collection when the amount of memory allocated since the last collection exceeds one third of the sum of twice the memory occupied by composite objects, the memory occupied by atomic objects, twice the stack size, and the root set size. The idea is two-fold: amortize the cost of collections, and keep the application's memory footprint low.

We added an adaptive scheduler. If the heap exceeds 15/16s of the available memory (the *threshold*), and the amount of unallocated memory exceeds a third of the available memory, we try to discard unused heap memory such that the heap uses less than 7/8s of the threshold. If the heap size still exceeds the threshold, we trigger a garbage collection and again try to discard unused memory such that the heap uses less than 7/8s of the threshold.

For our modifications to be effective, we also changed two functions to avoid reallocating discarded memory when there is still non-discarded memory available (namely, `GC_merge_unmapped` and `GC_allochblk_nth`). Although the library supports unmapping unused memory, this feature is not enabled by default and thus appears to have some bit rot. In this regard, these changes could be considered bugs in the original code base.

The Viengoos-specific changes were limited to just a single function, `GC_should_collect`, in which we implemented our scheduling policy. This was surprisingly non-invasive and increases our confidence that many programs and libraries can be easily adapted to use the type of interface that we propose, and that these changes will be integrated upstream even before the interface is widely accepted by the community.

5.1.1 Benchmark

We based our benchmark on the John Ellis and Pete Kovac garbage collection benchmark, which was written around 1997, and ported to the Boehm garbage collector by Hans Boehm.[1] The benchmark allocates two data structures, which remain live during the entire execution of the program. It then enters a loop and builds a number of binary trees of varying depths. After creating a tree, it immediately overwrites the pointer to the root node thereby making it available for collection.

We modified this benchmark in two ways. First, we loop over the tree creation loop 400 times. This lengthens the execution time, which allows us to better measure the effectiveness of the scheduler. Second, we introduced a memory hog. When enabled, approximately 45 seconds after the benchmark starts, it allocates 10 MB of memory per second until it has allocated half the initially available memory (in our case, approximately 820 MB). After sleeping for 90 seconds, it then deallocates the memory, again at a rate of 10 MB per second.

To estimate the possible throughput of the adaptive scheduler under a mature, highly-optimized system, we added a fixed scheduler. It acts like the adaptive scheduler except, it assumes that a fixed amount of memory is available. For our experiments, we fixed it to use 1.7 GB of memory. This is approximately the amount of memory available to the benchmark on Viengoos when there is no memory hog; when run on GNU/Linux without a memory hog, it does not thrash.

When the memory hog runs on GNU/Linux, we also lock the memory that it allocates. This is equivalent to the behavior of the memory hog on Viengoos, which only allocates its share of memory, and thus is not subject to paging. It also matches the intent: to model the memory use of an active program with a large working set. Alternatively, we could have had the memory hog enter a tight loop touching all of the memory to keep it in core. This, however, uses CPU time, which makes comparing throughput more difficult.

5.1.2 Results

Table 2 shows the number of garbage collections, the time spent in the garbage collector, and the execution time for each of the eight configurations. We first note that the modified schedulers significantly reduce the time spent in the collector. On both systems, when using the default scheduler, the benchmark spends more than 50% of its time collecting. This is reduced to less than 10% when using the adaptive or fixed scheduler without a memory hog. These savings translate to an overall decrease in execution time. We do not expect this speed-up to be typical—the benchmark specifically

[1] Available at `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/GCBench_OGC.cpp`.

	# GCs	GC Time Sec.	GC Time %	Time Sec.	Time Rel.
Vien., Adapt	134	10.2	3.6%	284.8	1.25
Vien., Adapt, Hog	223	13.4	4.4%	303.9	1.33
Vien., Def	43881	208.4	53.9%	389.0	1.71
Vien., Def, Hog	43881	214.7	53.6%	400.3	1.76
Linux, Fix	119	18.2	8.0%	228.0	1
Linux, Fix, Hog	119	50.4	12.6%	388.2	1.70
Linux, Def	42809	187.1	58.2%	321.4	1.41
Linux, Def, Hog	42809	187.3	58.9%	318.2	1.40

Table 2: Number of collections, time spent collecting, and time to execute for each configuration of the garbage collection benchmark.

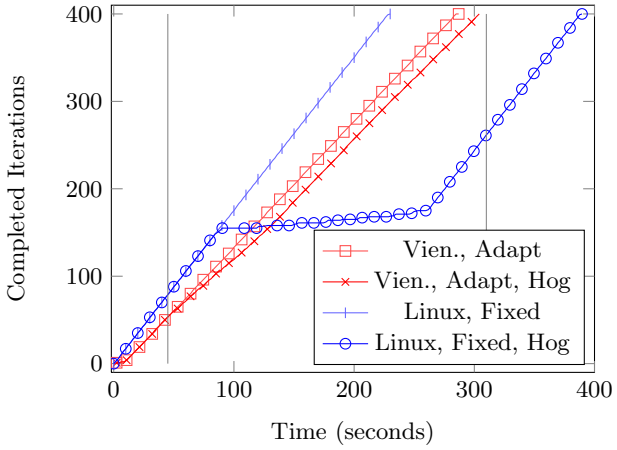

Figure 1: Time vs. iterations for the benchmark running on Viengoos and on GNU/Linux, using either the adaptive or fixed scheduler, and with or without a hog. Vertical lines show when the hog starts allocating memory and when it has released its memory.

exercises the collector and does little actual work—however, we suspect from [3, 32] that it will be significant.

The progress of the benchmark using the adaptive and fixed schedulers both with and without a memory hog is depicted in figure 1, which shows a plot of time vs. the number of completed iterations. A straight line corresponds to steady progress, and a steeper slope, to more work per unit time. On Viengoos, the presence or absence of the memory hog is hardly noticeable. This is also the case for the default scheduler. It, however, remains unaffected by the hog simply because it uses little memory; the cost is a 33% increase in execution time relative to the benchmark using the adaptive scheduler. When the fixed scheduler is used on GNU/Linux and there is a memory hog, the garbage collector makes essentially no progress. This is because while the memory hog is active, the garbage collector thrashes and makes very little progress. It is only because the memory hog releases its memory that the benchmark even completes in a reasonable time.

The adaptive scheduler's adaptation to the memory hog's presence is shown in figure 2, which plots time vs. heap size. We observe that as the memory hog allocates memory, the garbage collected program decreases its allocation accordingly. When the memory hog again frees the memory, the amount of available memory increases and the garbage collector makes immediate use of it.

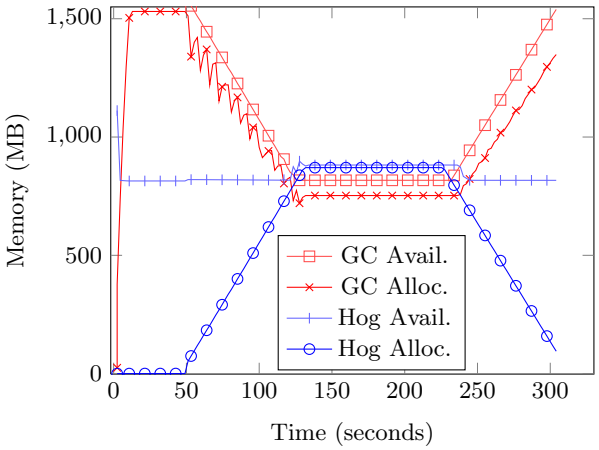

Figure 2: Time vs. memory, either allocated or available, for the benchmark and the memory hog running on Viengoos, using the adaptive scheduler.

When the memory hog has reached its pinnacle, it appears that the system reports that there is more memory available to it than its actual share. The reason for this is that the garbage collector maximally uses a bit less than is available to it, and when it approaches that mark, it frees memory until it is below a low-water mark. This free memory is reported as available to the memory hog. If the memory hog were to use this memory, it would *not* take away from the amount of available memory reported to the garbage collector.

5.1.3 Overhead

Table 2 shows that the benchmark using the adaptive scheduler on Viengoos runs 20% slower than the benchmark using the fixed scheduler on GNU/Linux, and using the default on Viengoos, about 15% slower than using the default scheduler on GNU/Linux. We do not view this as a flaw is our resource management approach but due to the fact that the system is not yet optimized and the use of the object capability model.

One example of an unoptimized component is the page ager. We measured that the the system spends 4% of its time in the page ager. One possible explanation is that the page ager samples access bits for every frame every iteration. A better approach would be to use exponential back off: if a frame has not been accessed, wait two iterations before checking it, if it has still not been accessed, wait four, etc.

The issue with the object capability model is that every object manipulation requires an IPC. In particular, this makes address space operations very expensive relative to the cost on GNU/Linux. For instance, allocating and installing a single page in the address space requires at least 2 IPCs. On GNU/Linux, the same can be done for large numbers of pages with just a single `mmap` invocation.

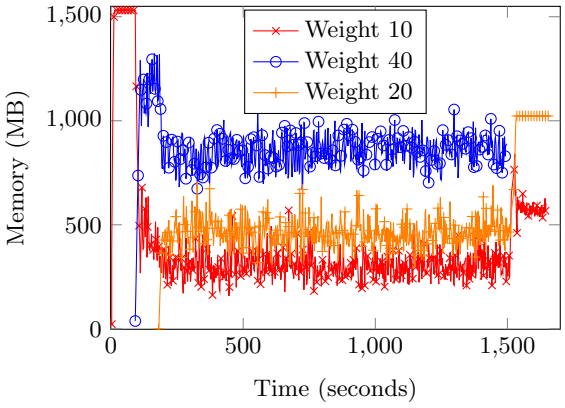

Figure 3: Time vs. memory for three garbage-collected applications with overlapping execution.

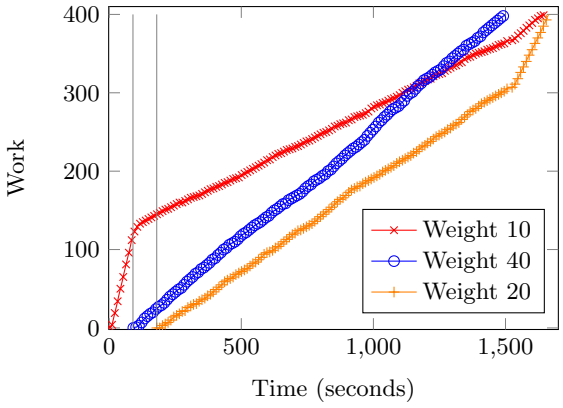

Figure 4: Time vs. amount of work completed for three garbage-collected applications with overlapping execution.

5.2 Multiple Adaptive Applications

To determine how well the framework supports multiple adaptive programs, we ran three instances of our GC benchmark. They were run at the same priority level but with weights 10, 40 and 20. (The weights only affect the instance's memory allocation: currently, Viengoos simply schedules the active threads' access to the CPU in a round-robin fashion.) Their starts were staggered: the second instance was started 90 seconds after the first and the last instance, 180 seconds.

Figure 3 shows a plot of time vs. the amount of memory allocated to each instance. When the system starts, the first instance immediately uses most of the memory. When the second instance starts 90 seconds later, the first instance adapts. Given their weights, the first instance should now use just a fifth of the total memory and the second, the remaining 80%. This is what happens: as the second grows to its 1.2 GB share, the first instance backs off to use about 400 MB of memory. When the third instance starts after 120 seconds, the first two instances adapt appropriately. Finally, as each instance completes, we observe similar behavior: the remaining instances adapt to use the newly available memory.

The allocations remain within approximately 50 MB of the expected allocations. The fluctuations are due in part to the fact that the garbage collector is designed to release memory to the system when it exceeds its threshold, which causes the amount of memory reported to the others to correspondingly increase.

Figure 4 shows how an instance's progress changes as more or less memory becomes available. Before another instance starts, the first instance completes a significant number of iterations per unit time. This drops noticeably when the second instance starts and the first instance's allocation decreases to just 20% of its prior allocation. This decreases again when the third instance starts. When the second instance, which uses approximately 800 MB finishes, both instances increase the number of iterations per unit time proportional to their new allocations.

5.3 Cache Management

To verify that availability information can help when managing a cache, we built a program that queries a database, and can cache the results. We measured the throughput for a number of fixed cache sizes as well as when the cache is managed by an adaptive policy based on the published availability. We also ran the last configuration in the presence of a memory hog.

5.3.1 Benchmark

The benchmark queries a database, which is managed by SQLite 3. It contains three temporary tables each with two columns, one containing a key and the other a value, and 1024 rows. A query consists of two selects: the first selects a row in the first table, and the second selects a row in the second table and joins the result's value on the third table's key. Each select yields one row. This is repeated 10 times for each query. Each resulting object is 5 MB. All bytes of the objects are written to.

The program queries the database $100,000$ times drawing from a pool of 1000 unique queries. We distribute the queries according to an exponential distribution. This reasonably approximates a long-tail distribution such as the Zipf distribution, which has been observed in practice. For instance, web pages accesses [22] and DNS lookups [8] are both known to conform to this distribution.

The fixed-sized cache was implemented using a simple algorithm: when the number of cached queries exceeds the cache size, discard 20% of the cache in a least-recently-used fashion. This is maintained by keeping all entries on a doubly linked list sorted by access time. When an entry is accessed, it is moved to the head of the list; when evicting entries, the required number is taken from the end of the list.

A similar approach is taken to managing the cache in the adaptive case: when there is pressure, we release enough memory such that the program only uses 7/8s of the available memory.

When enabled, the memory hog begins 45 seconds after the

	Hits		Time	
	Hits	%	Sec.	Rel.
348 lines	92300	92.3%	80.1	1
256 lines	82595	82.6%	144.4	1.80
64 lines	27646	27.6%	501.3	6.26
16 lines	7099	7.1%	599.7	7.49
4 lines	1962	1.9%	629.4	7.86
Adaptive	90574	90.6%	84.3	1.05
Adaptive, Hog	78230	78.2%	284.6	3.55

Table 3: The number of cache hits and the total execution time for each configuration.

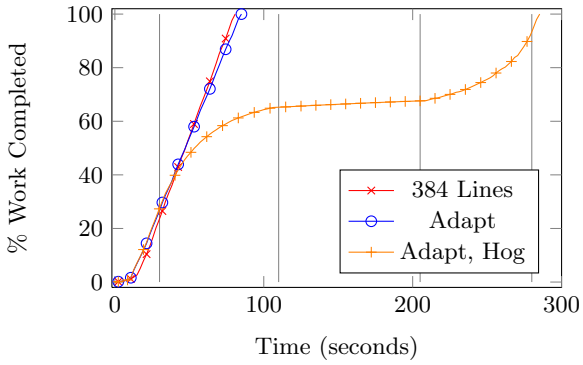

Figure 5: Time vs. the word.

benchmark starts, it allocates memory, 20 MB per second, until it has allocated 95% of the memory in the system. After about a minute, it deallocates the memory, again, at a rate of approximately 20 MB per second.

5.3.2 Results

Table 3 shows the results for running the benchmark in several configurations. We first observe that as the cache size increases, the hit rate increases and the throughput increases. When using most of the system's memory, the cache covers approximately 20% of the objects and results in over a 90% hit rate. Figure 5 shows the amount of work completed per unit time. When using the adaptive cache and there is no competition for memory, it closely tracks the performance of the large fixed-sized cache. When there is memory pressure, however, it adapts. Its performance, however, remains proportional to the amount of memory available to it. This change in availability is shown in Figure 6.

The effectiveness of caching is primarily dependent on the expected access frequency and the cost of maintaining the cache relative to the cost of computing the objects. As the cost of computing objects increases, the importance of a cache grows. We expect that using an adaptive cache will help in many situations and that the benefits will be proportional to these results. As our prototype becomes increasingly functional, we plan to experiment with more realistic workloads to confirm this.

5.4 Shared Memory

To test the effectiveness of using access frequency to distribute the cost of shared memory, we designed a test in which multiple activities use a block of shared memory.

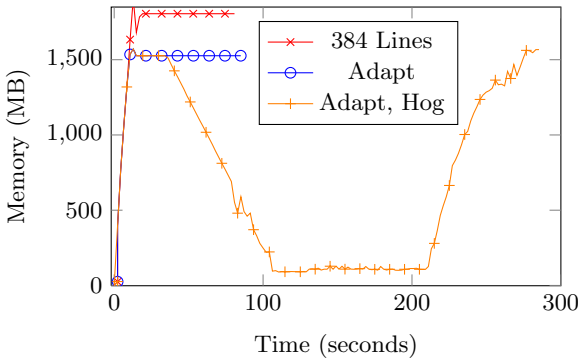

Figure 6: Time vs. memory use.

The test allocates a 4 MB block of memory (1024 pages). It starts three threads, each using a separate activity. Each thread executes a loop. During each iteration, a thread accesses a random number of the shared pages according to an exponential distribution. The pages are selected randomly according to a uniform distribution. After accessing the pages, the thread sleeps for 10 ms. We periodically record how many pages are charged to each activity. This includes the data and code that the threads use, which consists of an additional 55 pages, and is also mostly shared.

Figure 7 shows the number of frames accounted to each of the three activities. In this case, they show that the shared memory accounting mechanism is relatively accurate.

We ran the benchmark again without the pausing between accessing pages. In this case, each activity is correctly accounted a third of the memory. However, this is only true over a long period of time: for each sample, almost always, a single activity is accounted all of the memory. This is because when a thread runs, it accesses all pages in the buffer during its time slice. Thus, the first thread to be scheduled after the shared memory has been unmapped claims all the memory.

We do not view this as problematic: we suspect that it is rare that a single buffer that is shared and accessed in its entirety needs to be correctly accounted over short periods of time; we suspect that long-term fairness is more important. An interesting pathological case is when the size of the shared buffer approximates or exceeds the amount of memory available to any single principal using it. In this case, it is possible that one principal claims all the memory and then, due to memory pressure, is selected to yield some memory. Some of that memory may be scheduled for eviction. However, as it is not immediately discarded, when the next activity uses the buffer, it will claim the memory.

In practice, we suspect that the main use of shared memory will be shared binaries and data. Some code, such as the string routines provided by the standard C library, will be used often and by many. The problem then is that as we unmap shared objects to detect other users, using a shared object introduces additional soft page faults.

Our current strategy for determining the use of a shared

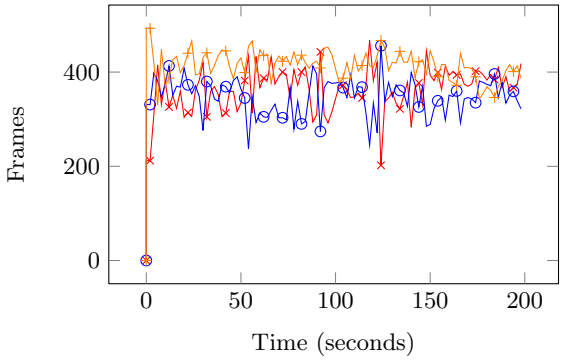

Figure 7: Time vs. the number of frames accounted to an activity.

	Iterations	Time (μs)	Iter. / μs
Unmap			
Thread	210,144,069	198,468,750	1.0588
Thread	210,292,589	198,500,000	1.0594
Thread	210,377,718	198,484,375	1.0599
Total	630,814,376		1.0594
No unmap			
Thread	210,590,608	198,468,750	1.0611
Thread	210,746,341	198,500,000	1.0620
Thread	210,857,024	198,484,375	1.0623
Total	632,193,973		1.0618

Table 4: The number of iterations per $\mu-$second when shared memory is either periodically unmapped or not.

page is to simply tear down all mappings. A better approach would be to mark the PTEs corresponding to a shared page as invalid, and, mark them all as valid after the page is again claimed. This eliminates any tear down and reinitialization costs and reduces the number of soft page faults.

To determine the cost of accounting the shared memory, i.e., the cost of the soft page faults and the additional claiming, we ran the previous experiment two more times. This time, we again did not include a pause. Also, for one run, we disabled unmapping shared pages. Each instance ran for just under 200 seconds. Table 4 shows the number of iterations of the main loop for each thread, the total time each thread ran and the average iterations per microsecond. The time measurement has an accuracy of plus or minus 10 milliseconds. When not unmapping memory, we observe a 0.23% increase in throughput. In this experiment, we only used 4 MB of memory, which was unmapped every 2 seconds. If more memory is shared, the costs will grow proportionally.

6. RELATED WORK

Waldspurger presents a comprehensive treatment on using proportional share mechanisms to schedule resources in [30]. He reifies resource rights by way of so-called tickets, which a principal uses or delegates to other computations (i.e., other principals) that it wishes to fund. This abstraction allows a single computation to be funded by multiple principals. His focus is primarily on scheduling CPU time, however, he also briefly considers memory. He notes that memory is unlike CPU in that its management is based on revocation and not allocation and thus traditional scheduling algorithms are not appropriate. He proposes a *minimum-funding revocation scheduler*: when memory needs to be reclaimed, the memory manager selects the principal that provides the fewest number of tickers per frame. He presents results from a simple simulation; no implementation is provided, however. Waldspurger also does not consider feedback in this work nor does he discuss how to account shared memory.

In [31], Waldspruger presents the balloon driver, a simple mechanism allows the VMM to dynamically reallocate memory among guests by coercing unmodified guest operating systems to adapt to their new allocation. The idea is simple: the balloon driver is an operating-system-specific driver that communicates with the virtual machine monitor (VMM) through a private channel. It allocates and releases operating system's memory according to the VMM's instructions. These allocations cause the operating system to use its normal adaptation mechanisms, which exploit operating-system-specific knowledge, to make memory available for the driver. Ignoring trust issues, we are interested in pushing these types of interactions down another layer and exploiting application-specific knowledge. As applications are the direct users of memory, they have significantly more semantic knowledge of how it is used.

Alonso and Appel have garbage collectors provide their collection parameters directly to the memory manager, which responds with how the application should size its heap [1]. This is not a general solution. Indeed, it is not even a solution for managed run-times as it assumes a particular collector implementation. Also, it is designed to globally minimize collector overhead; it does not consider preferences.

Yang et al. argue that GC applications perform better when they use all available memory [32]. They observe that to do this in a dynamic environment requires feedback to avoid thrashing. The paper makes two contributions. First, the authors describe how to estimate the amount of memory required to perform a collection using a copy collector. Second, they describe an availability metric: 'process's current allocation' plus 'free/unused system memory'. They demonstrate that their approach works well for a single adaptive program. They also run a benchmark that shows that multiple adaptive applications can run on their system, however, they lack a mechanism to effectively distribute free memory, which they note is a subject for future work.

Iyer et al. present a so-called severity metric, which summarizes memory pressure and the cost of paging [21, 20]. If severity is low, there is free memory and applications may grow; if it is medium or high, applications should shrink. The authors recommend that applications grow or shrink by tens of MBs of memory per second until the severity metric reports normal. As the severity metric summarizes the cost of paging, an application can determine whether to free memory given the cost to regenerate the content. To distribute memory among multiple applications, the severity metric is scaled by the application's 'nice' value. This scaling is contrary to the metric's intended use, which is to determine whether to free memory or to have it paged.

Significant work has been done exploring how to allow applications to better manage the resources available to them.

V++ [17], exokernel [13], Nemesis [15] and SawMill [4] export physical resources and use visible revocation to allow programs to use application-specific knowledge to manage their resources. The question of how to distribute resources among multiple computations is unanswered by exokernel and Nemesis. Indeed, the exokernel architects state that the appropriate policies are "determined more by the environment than by the operating system architecture" [13].

On V++, memory is distributed using a market-based approach [16]. The system page-cache manager maintains a bank account for each top-level principal. Sub-accounts can be created yielding a hierarchy of accounts. Memory is leased for a certain amount of time. There are three types of lease requests: high priority, normal priority and low priority. The cost of each type of lease is fixed. A high-priority lease costs more than a normal priority lease; and, a low-priority lease does not cost anything but may be terminated early. This fixed-cost scheme was chosen as variable prices appeared too hard for programmers to reason about. Their scheme does not consider shared memory.

SawMill [4] tries to push policy decisions as close to applications as possible. To this end, it uses a distributed approach to managing memory: physical resources are distributed and visible revocation is used to revoke them. The authors do not describe how resources can be efficiently allocated among multiple children or good policies to reclaim them. This necessarily imposes some policy. In a general-purpose system, some *de facto* policy would develop simply because multiple applications must negotiate resource allocations. Also, as applications will often have memory from multiple sources, e.g., an anonymous memory server and a file server, and revocation is done hierarchically, the application will be limited in what it can revoke to the memory that it received from the provider requesting memory be returned, i.e., memory becomes less fungible thereby reducing flexibility.

7. CONCLUSIONS

General-purpose operating systems do not support memory-adaptive applications. As we have shown, memory adaptations can significantly increase the performance of programs using garbage collection and caches of computed data. These adaptations are becoming more important given trends to increase use of consolidation and the wider range of configurations that users expect their applications to run on, e.g., smart phones.

This paper makes three contributions. We have presented the design and implementation of a minimum-funding revocation memory scheduler. We have shown how to compute how much memory is available to each resource principal thereby enabling informed and timely adaptations. We have also presented a mechanism to account shared memory based on access frequency. Our benchmarks show that the algorithms are effective for controlling adaptations in the face of changing resource availability. Moreover, they are effective at allocating memory among multiple aggressively-adaptive memory applications.

8. REFERENCES

[1] R. Alonso and A. W. Appel. An advisor for flexible working sets. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 153–162, 1990.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM SOSP*, pages 95–109, 1991.

[3] A. W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[4] M. Aron, L. Deller, K. Elphinstone, T. Jaeger, J. Liedtke, and Y. Park. The SawMill framework for virtual memory diversity. In *8th Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, Jan. 2001.

[5] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX OSDI*, Feb. 1999.

[6] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[7] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[8] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of Infocom 1999*, pages 126–134, 1999.

[9] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the adms query optimizer: integrating query result caching and matching. In *EDBT '94*, pages 323–336, 1994.

[10] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Trans. Netw.*, 5(6):835–846, 1997.

[11] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[12] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, Mar. 1966.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *15th ACM SOSP*, pages 251–266, Dec. 1995.

[14] C. Fleck. What's the coolest app that doesn't work on the iPhone .... yet ? `http://community.citrix.com/pages/viewpage.action?pageId=51937665`, Dec. 20, 2008.

[15] S. M. Hand. Self-paging in the nemesis operating system. In *3rd OSDI*, pages 73–86, 1999.

[16] K. Harty and D. Cheriton. *A Market Based Approach to Operating System Memory Allocation*, pages 126–155. World Scientific Publishing, River Edge, New Jersey, 1996.

[17] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *ASPLOS-V*, Oct. 1992.

[18] K. Hazelwood and M. D. Smith. Managing bounded code caches in dynamic binary optimization systems. *ACM Trans. Archit. Code Optim.*, 3(3):263–294, 2006.

[19] D. Hull, W. Feng, and J. W. S. Liu. Operating system support for imprecise computation. In *AAAI Fall Symposium on Flexible Computation*, Nov. 1996.

[20] S. Iyer. *Advanced memory management and disk scheduling techniques for general-purpose operating systems.* PhD thesis, Rice University, Houston, Texas,

November 2005.

[21] S. Iyer, J. Navarro, and P. Druschel. Application-assisted physical memory management. Technical report, Rice University, 2004.

[22] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. Dns performance and the effectiveness of caching. *IEEE/ACM Trans. Netw.*, 10(5):589–603, 2002.

[23] H. Levy and P. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Computer*, 15(3):35–41, 1982.

[24] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: creating a "functional" internet. In *EuroSys 2007*, pages 101–114, Mar. 2007.

[25] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, 1994.

[26] S. Peter, A. Baumann, T. Roscoe, P. Barham, and R. Isaacs. 30 seconds is not enough! A study of operating system timer usage. In *EuroSys 2008*, April 2008.

[27] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge, Aug. 1995.

[28] J. S. Shapiro and N. Hardy. EROS: A principle-driven operating system from the ground up. *IEEE Software*, 19(1):26–33, 2002.

[29] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[30] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-share Resource Management*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[31] C. A. Waldspurger. Memory resource management in VMware ESX server. In *5th OSDI*, Dec. 2002.

[32] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *7th OSDI*, Nov. 2006.