Submitted to EuroSys 2009.

Viengoos: A Framework for Stakeholder-Directed Resource Allocation

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

Abstract

General-purpose operating systems not only fail to provide adaptive applications the information they need to intelligently adapt, but also schedule resources in such a way that were applications to aggressively adapt, resources would be inappropriately scheduled. The problem is that these systems use demand as the primary indicator of utility, which is a poor indicator of utility for adaptive applications.

We present a resource management framework appropriate for traditional as well as adaptive applications. The primary difference from current schedulers is the use of stakeholder preferences in addition to demand. We also show how to revoke memory, compute the amount of memory available to each principal, and account shared memory. Finally, we introduce a prototype system, Viengoos, and present some benchmarks that demonstrate that it can efficiently support multiple aggressively adaptive applications simultaneously.

Categories and Subject Descriptors D.4.10.a [*Operating Systems*]: Support for Adaptation

Keywords Resource Management, Accounting, Adaptive Applications, Utility, Stakeholder Directed

1. Introduction

General-purpose operating systems schedule resources based on the principles of the working set model of program behavior, namely, that a program exhibits locality of reference and its demand is independent of its environment (7). For programs conforming to this model, resources can be efficiently scheduled by observing memory accesses and ensuring that active programs have their recently-used memory in core. This is the case, for instance, for a text-book implementation of quick sort: its resource requirements are primarily determined by its input. Providing it with more memory than the working set model of program behavior determines is appropriate does not improve its performance; providing it with less simply delays its completion.

Many programs could improve both their performance as well as the system's by adapting to their environment. These programs could select among multiple algorithms or vary algorithmic parameters according to resource availability so as to improve some performance metric. Such programs include those that use a garbage collector (3; 1; 24), those that could use a cache, and those that could reduce quality to better ensure some timeliness property (6; 13).

Most programs which could adapt do not. This is because a program can only gain useful knowledge of its environment using ad-hoc heuristics, which tend to be difficult to implement and sufficiently unreliable that not adapting is not only simpler but also more efficient.

The inability to effectively adapt is becoming more acute by the trend to use the same software on an increasingly-wide spectrum of hardware—from smart phones and internet tablets to desktops and servers. For such a wide range of configurations, statically determining the right trade-offs, e.g., choosing an appropriate cache size at compile time, results in generally under- or over-utilizing resources.

Enabling effective adaptation on general-purpose operating systems is not just a question of providing programs with information regarding the amount of available resources: general-purpose operating systems determine availability based on observed demand, however, demand is exactly what adaptive applications vary according to availability. This is illustrated in figure 1. The result is a positive-feedback loop, and, ultimately, ineffective scheduling.

As the purpose of adapting is to increase a program's utility, we propose a stakeholder-directed resource allocation scheme in which a computation's stakeholders influence how resources are allocated.

A computation's stakeholders are those agents that have an interest in its execution. We observe that a computation typically only directly represents positive utility for a single stakeholder: the agent that started it. By allocating a computation's resources out of its parent's allocation and allowing the parent to control how the resources allocated to it are

$$\begin{tikzcd}
\text{Kernel} \\
f(\mathit{demand}) = \mathit{availability} \arrow[d, bend left, "\mathit{availability}"] \\
f(\mathit{availability}) = \mathit{demand} \arrow[u, bend left, "\mathit{demand}"] \\
\text{Adaptive Application}
\end{tikzcd}$$

Figure 1. Adaptive applications create a positive-feedback loop when managed according to working set principles. Although the system may stabilize, no mechanism ensures that it converges to a configuration with maximum utility.

$$\begin{tikzcd}
 & \text{Kernel} \\
 & f(\mathit{parameters}, \mathit{demand}) = \mathit{availability} \arrow[d, shift left, "\mathit{availability}"] \\
f(\mathit{output}) = \mathit{parameters} \ \text{Principal} \arrow[ur, "\mathit{parameters}"] & \\
 & f(\mathit{availability}) = (\mathit{demand}, \mathit{output}) \ \text{Computation} \arrow[uu, shift left, "\mathit{demand}"] \arrow[ul, "\mathit{output}"]
\end{tikzcd}$$

Figure 2. Scheduling feedback loop where the parent-supplied scheduling parameters in addition to demand determine the schedule.

distributed among its children, two properties emerge: a parent is responsible for its children's allocations, and computations form a recursive hierarchy with the system administrator at the root. Within this framework, any negative utility that a computation attributes to another must be attributed to a common ancestor, which made the explicit decision to allocate its resources the way it did. Thus, assuming rational agents, the schedule will converge towards one with maximal expected utility. The remaining difficulty then is enabling agents to easily express their preferences.

A naïve approach would have parents provide the parameters to a multi-variable utility function for each of their children. This approach is not usable: users and developers lack the required information. Our framework uses parent-assigned, coarse-grained parameters. The resource manager uses these as well as demand to distribute resources. Agents then tweak these parameters based on high-level feedback. This results in a negative-feedback loop, shown in figure 2.

Contributions: This paper makes the following contributions: it presents a framework for stakeholder-directed resource allocation; it describes how to revoke memory within such a framework, how to calculate the amount of memory available to each computation, and how to account shared memory; and, it presents a prototype system, Viengoos, that demonstrates that the framework efficiently supports the running of multiple adaptive applications simultaneously.

2. Analysis

The transparent multiplexing of storage is a valuable tool in helping to manage complexity (8). This multiplexing can be efficiently scheduled when the managed programs conform to the working set model of program behavior, which models program behavior as exhibiting locality of reference and having demand that is primarily a function of its input (7).

Adaptive programs do not conform to this model: their demand is a function of the available resources. Moreover, the resources they use are not required for progress: if there is pressure, they may be forgone. For instance, a garbage collector can vary when collections occur; caches can be sized according to availability; and, a more appropriate algorithm or algorithmic parameter, e.g., quality, can be selected.

Supporting adaptive applications does not necessarily require exporting physical resources and thus forgoing the advantages of virtual memory: how much memory is available often suffices. A garbage collector can time collections so that they occur just before paging would be required; caches can be sized (and resized) according to the amount of available memory; and, choosing among alternative algorithms or selecting the most appropriate parameters becomes a straightforward exercise. Moreover, the possible side-effect of using more resources—that contention and thus the chance of thrashing increases—would disappear as an application would know when this is about to occur and could proactively scale its use.

Computing availability within the context of current resource managers is not enough to efficiently support adaptive applications: their behavior would create a positive-feedback loop. This arises as their demand is a function of availability while the resource manager distributes resources according to demand. This is illustrated in figure 1.

The question then becomes how to distribute resources given that multiple applications would like to adapt. The answer suggests itself: as the motivation behind adaptation is to maximize expected utility, resources should be distributed so as to maximize the expected utility of the stakeholders.

2.1 Describing Utility

Determining the expected utility that a computation has for a stakeholder poses a difficult problem: utility is a subjective measure. Traditionally, general-purpose resource managers have avoided giving too much control to user programs as this would have provided malicious or buggy programs significant control of the system. More control is only a problem if it can be used to create destructive interference (20): user programs can be relied upon if the quality of the provided information yields, from their perspective, qualitatively similar results. That is, providing good information must correspond to better performance for the agent itself and bad information to worse performance.

The problem then is finding an appropriate language for describing expected utility. The resource manager understands low-level resources such as CPU cycles and frames of memory. Stakeholders, however, are interested in high-level metrics such as frames per second and cannot easily express these requirements in terms of the low-level resources that the resource manager understands. This is not only because there are complex data dependencies but also because the computations that an agent is interested in are often opaque to it: users and developers have at best a rough notion of the resource requirements of the programs and libraries they use; such details are rarely exposed.

A good solution is not one that requires users and programmers to reformulate their high-level requirements in terms of low-level resources: this makes the system difficult to use, decreasing its reliability. Instead, the disconnect needs to be bridged in a way that allows them to simply and directly use the metrics they are already interested in.

2.2 Accounting

When distributing resources according to a non-trivial policy, the relevant resources must be correctly accounted. If memory is not correctly accounted, then determining how much a principal uses or is available is less accurate and less useful. This is often the case for kernel resources.

A mechanism must also be provided to usefully account shared resources. Memory, for instance, is shared in the form of program text. Other uses of shared memory are common and will increase if the principal granularity is increased.

2.3 Dynamic Reallocation

We are specifically interested in better scheduling for dynamic, multi-programmed systems, that is, those environments in which there are multiple programs running simultaneously with variable demand, which is characterized by resource-intense spikes and periods of relative quiescence. Such systems include PDA-like devices, desktops, and multi-service servers. On the first two types of systems, when a user is interacting with a program, it often exhibits increased demand relative to when it is in the background, most of which are at any given time. On a web server, one site may experience a surge in activity, however, it is unlikely that this will happensites that it serves at once.

A concern with such systems is the effective utilization of resources. This means that if a program requires more resources than are available to it and other programs are not using all of the resources available to them, it ought to be allocated the idle resources—at least until the other programs require those resources. That is, the desire is that the scheduler be work conserving and dynamically reallocate resources.

Such a scheduler is poor for hard-real-time applications: although reclaiming resources may be prompt, it may introduce non-trivial latencies. It is also poor for very high-security systems: a work-conserving scheduler that exposes availability introduces a high-bandwidth covert channel.

3. Stakeholder-Directed Management

We now present a framework in which resources are distributed based on stakeholder preferences with the result that expected utility is maximized, and in which a program can easily determine the amount of memory available to it.

3.1 Maximizing Utility

An action's utility is a subjective measure that is determined by its stakeholders. An action's stakeholders are those principals that are affected by the action in either a positive or a negative manner. In the case of resource management, we are interested in the actions related to resource allocation.

Providing a larger allocation to a computation represents a non-negative utility to it as well as any principal that has an interest in its performance. Such an allocation corresponds to a negative utility for any principal that as a result receives a smaller allocation but which could have improved its performance given a larger allocation. It is also a negative utility for any principal that has an interest in such principals.

The question then is how to combine these conflicting preferences. We observe that no program runs for its own sake, and that a program only directly serves the ends of the computation that started it. A web browser, for instance, only directly serves the ends of the user on whose behalf it runs. For a principal that is not a direct ancestor, resources allocated to the web browser that it or a computation that serves its ends could have used represent negative utility.

Because a computation only directly serves the ends of the principal that started it, it can be logically considered a part of that principal: ignoring protection issues, the principal could have executed the computation itself. As this is the case for all computations, a recursive hierarchy emerges with the system administrator at the root.

By then allocating resources to a principal and requiring that it controls how those resources are allocated among its child computations, expected utility is necessarily globally maximized. This is because a computation's resources are provided by its sole positive stakeholder. Thus, any negative utility must be attributed to the nearest common ancestor. Since it explicitly allocated its resources in the way it did (so as to maximize its expected utility), the fact that a computation received less than it desired is consistent with globally maximizing expected utility. Consider a user with two programs each of which would like 80% of the user's allocation. Maximizing expected utility means respecting the user's interests, not the programs, which only serve the user.

This approach is also resilient to destructive interference: when a principal pays for the resources that computations executing on its behalf use, the principal has a strong incentive to ensure that they are used efficiently and distributed appropriately; it cannot cheat the system.

Shared servers do not necessarily violate this compositional model. When a server executes some request on behalf of a client, generally, that request does not represent positive utility for another client. One reason for this is because such servers are primarily concerned with decomposing a larger object, e.g., a file system may decompose a disk partition, and thus most of its work is traversing meta-data. The client can then be viewed as executing in the context of the server, and should use its own resources.

3.2 Allocation Decisions

There are two main ways to allow a principal to control how its resources are distributed among its children. First, it can do it itself by interposing on all allocations and revocations. This is the approach SawMill takes (4). Alternatively, it can describe the policy and have a third party act on its behalf.

The claimed advantage of the first approach is that agents are able to implement any policy they like; there is no system-imposed language in which they must, perhaps clumsily, express their intents. This appears to provide fine-grained control over how resources are managed. The alleged control is an illusion: because multiple agents whose development is not coordinated must communicate, a way to negotiate resources must be agreed upon. The result is that some *de facto* language will develop with no advantages over a centrally imposed language. Moreover, this approach has additional costs: interposition introduces overhead, and, as resource use is negotiated pairwise, local changes in demand will take longer to propagate through the system than when using a more agile, central manager.

A centralized approach has an important advantage: it does not need to support visible revocation (10), which is required to support recursive revocation. The problem with visible revocation is that it is a potential source of destructive interference, which can only be tamed by way of timeouts. This introduces a perverse incentive: agents will keep resources as long as allowed rather than revoking them promptly. It also adds the requirement that revocation be real-time capable. This property will deter experimentation, and make debugging more difficult as bugs may be triggered by difficult-to-reproduce load factors.

Further, hierarchical revocation results in the situation where an application receives a revocation request from a server and must release memory that it got from that server, even though, memory is fungible and any memory would do.

3.3 Scheduling Parameters

Using utility functions is problematic: whereas a resource manager deals with basic, low-level resources such as frames and CPU time, applications are interested in high-level metrics such as the time to respond to input, and neither can the resource manager easily measure higher-level metrics nor can a principal easily express its high-level metrics in terms of low-level resources as it usually does not know how the computations it is interested in are implemented.

There is, however, no requirement to use utility functions; the goal is to maximize expected utility. This can by done by way of a feedback loop that, via an iterative process, causes the scheduler to converge to an optimum. One such feedback loop was shown in figure 2: the resource manager provides knobs, which principals tune based on higher-level feedback.

The parameters that determine the resource distribution policy should be sufficiently expressive and flexible to cover the majority of the relevant scenarios. This must be tempered by complexity: the policies should be as simply as possible to ensure that the intent is easily understood. We propose two scheduling parameters, a priority and a weight. Allocation should not be based solely on these parameters, however: a principal's demand should be considered to allow easy identification of slack resources, and its working set should be taken into account to avoid thrashing. Because it is useful to allow principals with children to also allocate resources, child-relative parameters can be added. The presented algorithms do not include these but they are easily incorporated.

Ultimately, the allocation of resources among computations depends on some human's priorities and having a user change even something as simple as an application's priority is cumbersome. Fortunately, for interactive systems, utility can often be inferred by the window manager based on user interactions. If this is inadequate, a small slider managed by the window manager can be placed in each program's title bar as done in Nemesis. In other cases, developers and distributors can provide scheduling hints that can be easily confirmed by the user during installation, e.g., "audio player."

Priority A priority parameter allows a principal to determine the order in which its children have access to its resources: higher-priority children have the right of first refusal and can preempt lower-priority children.

This policy allows a principal to ensure that some child computations always have the resources they require if the principal itself has sufficient resources. A window manager would use this to ensure that the user stays in control: by giving the input handler a high priority, it can ensure that if the user wants to terminate some program, he or she can do so even if that program is consuming a lot of resources.

This policy is also useful for controlling low-priority principals such as cache managers and background applications, e.g., a file indexer. By assigning such computations a lower-priority, the parent is sure that they only have access to resources that are otherwise unused and unneeded.

Weight A weight parameter is one way to have multiple children at same priority level. Using a weighted-proportional share scheduler, a principal with weight 5 is entitled to 5 times the resources of a principal with weight 1.

This can be used to approximate the importance of principals that execute simultaneously: an important principal is given a larger weight with the result that it is entitled to more resources and can provide better preference. The weight of the program instance with the focus, for instance, may be increased, and that of minimized programs decreased.

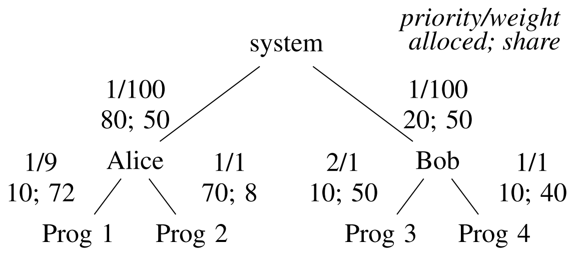

Figure 3. Scheduling parameters and possible allocations and shares. Alice and Bob have the same priority and same weight and thus are entitled to the same amount of resources. Alice's program 1 and program 2 have the same priority, however, the former is entitled to 9 times as many resources as the latter. Bob's program 3 has a higher priority than his program 4 and thus its demand preempts program 4's. Bob (and Bob's children) can preempt Alice's resources insofar as she is using resources to which Bob is entitled.

3.4 Memory Revocation

When there is contention, one or more principals must be found to preempt resources from. We search from the principals that most exceed their fair share according their respective scheduling parameters and working set size.

Algorithm 1 shows our approach. It starts by considering the root's children. It sets the working set factor to 1, meaning that it does not consider recently used (active) frames as allocated. It then iterates over each priority class and selects the principal such that its claimed frames minus its active frames corrected for by the working set factor scaled by its weight is greatest. If this is zero or less, this is repeated for the next-highest-priority class. If no principals remain, it doubles the working set factor (giving less weight to active frames) and repeats. Otherwise, it has found a victim. If that principal has no children, the algorithm has found the best victim. Otherwise, it recurses.

Using figure 3 to illustrate, the scheduler starts at the root and selects Alice: her and Bob's priority and weight are equal meaning they are each entitled to 50% of their parent's resources; as Alice has allocated 80% and Bob 20%, Alice has most exceeded her share. This process is then repeated for Alice. This time, her Program 2 is selected. Since it has no children, memory is revoked from it.

3.5 Computing Memory Availability

The recursive hierarchy can be used to compute the amount of memory that is available to a principal. A way to do this is shown in algorithm 2. The idea is to traverse the hierarchy and set each principal's availability to the maximum of its share and its allocated memory and to add to that the amount of free memory. We also adjust for local pressure, e.g., when a trend suggests that the principal will soon have less available memory, thereby better enabling proactive adaptation.

As other principals may have allocated more than their share, part of this memory could be "stolen" by a principal.

Algorithm 1 Selecting a principal to revoke resources from.

```
1:  function SELECTVICTIM($principal$)
2:      $ws\_factor \leftarrow 1$
3:      loop
4:          for $P \in \{$ $children(principal)$ grouped by
                priority, lowest to highest $\}$ do
5:              $excess \leftarrow 0$
6:              for $p \in P$ do
7:                  $t \leftarrow (p.alloced - p.working\_set$
                        $/ws\_factor)/p.weight$
8:                  if $t > excess$ then
9:                      $excess \leftarrow t$
10:                     $victim \leftarrow p$
11:             if $victim \neq nil$ then    ▷ If we have a victim.
12:                 return $SelectVictim(victim)$
13:         $ws\_factor \leftarrow ws\_factor \cdot 2$
```

Algorithm 2 Computing the amount of available memory.

```
1:  procedure AVAILABLE($principal$, $mem$)
2:      $principal.available \leftarrow mem$
3:      for $P \in \{$ $children(principal)$ grouped by priority,
            highest to lowest $\}$ do
4:          $alloced \leftarrow weight \leftarrow 0$
5:          for $p \in P$ do
6:              $alloced \leftarrow alloced + p.alloced$
7:              $weight \leftarrow weight + p.weight$
8:          $extra \leftarrow 0$
9:          for $p \in P$ do
10:             $share \leftarrow mem \cdot \frac{p.weight}{weight}$
11:             if $p.alloced > share$ then
12:                 $extra \leftarrow extra + p.alloced - share$
13:         for $p \in P$ do
14:             $share \leftarrow mem \cdot \frac{p.weight}{weight}$
15:             $entitle \leftarrow max\,(p.alloced, share)$
16:             ▷ Approximate extra frames $p$ could steal.
17:             if $p.alloced > share$ then
18:                 $steal \leftarrow \left(extra \cdot \frac{p.weight}{weight}\right) -$
                        $(p.alloced - share)$
19:             else
20:                 $steal \leftarrow extra \cdot \frac{p.weight}{weight}$
21:             $free \leftarrow mem - alloced$
22:             Available ($p$,
                    $(entitle + steal + free) \cdot p.pressure)$
23:         $mem \leftarrow mem - alloced$
```

For instance, consider three activities with the same weight, one of which has allocated all of the memory and the other two, none. When just considering the share, the available memory for the latter two are 1/3. If one of the principals allocates its share, the new configuration is 2/3s, 1/3 and 0. There is an implicit first-come first-served policy. If would be better if the unused memory entitled to the third principal is distributed according to the other principals' weights, which would result in each having half the resources.

The algorithm to calculate the amount that can be stolen only approximates the correct amount. The problem is that largely different weights will result in increasingly incorrect results because we scale the extra memory according to the total weight. This tradeoff avoids having to iterate over all the principals in the priority group and calculating how much each can steal from the others based on their respective weights, which is quadratic in principals.

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.

Availability information should be updated when availability changes. A naïve approach updates it periodically. A more intelligent one detects when availability has significantly changed and only then recomputes it.

3.6 Usage

For a principal to control the allocation policy of a computation, before starting the computation, it creates a principal object, sets a policy, and then delegates the child access to it. By designing the system such that resources are accounted to the principal that uses them, the principal can be sure that the computation is constrained by its policy.

When a principal uses server-implemented functionality, it again creates a principal object and sets a policy. It then invokes the server and passes a reference to the principal. The server can use it to allocate resources on behalf of the client, which are scheduled according to the client's policy. As the client controls the schedule and can revoke resources at any time, the server needs to be careful to avoid allowing this to result in destructive interference. This control is necessary to allow safe interaction with untrusted servers. Handling this requires careful data structure and interface design. Bounded timeouts, which guarantee the client promptness but enable server recovery, can also help.

3.7 Allocating Resources

There are two general ways to allocate resources. Either a principal can request a schedule before starting a computation, and the resource manager can admit it or not, or a principal can request resources lazily, i.e., on demand.

Using the former approach, a program requests a schedule before starting the computation. A more flexible variant is that a program submits an array of acceptable schedules, and the resource manager chooses the highest-ranked schedule that does not violate its policy (17). This requires being able to calculate *a priori* the required resources. We argued in the context of describing utility functions that this is hard.

With the latter approach, a program allocates resources while running a computation and at the point that it needs them. The system is also able to reclaim resources at any time, increasing its flexibility. This is how resources are usually scheduled on Unix-like systems.

4. Implementation

We implemented Viengoos on top of the L4Ka::Pistachio microkernel (23). We used Pistachio as a convenient hardware abstraction; we specifically avoided any advanced features that would have violated Viengoos's design goals.

Viengoos is an object-capability microkernel (9; 22). In an object-capability system, operations are modelled as method invocations on an object. Objects are designated by capabilities, which are protected references. Capabilities include their naming context, thereby avoiding the confusion arising from symbolic names (21).

Viengoos is designed to be resilient to destructive interference (20), to thoroughly account accounted resources, to avoid kernel allocations, to export atomic, restartable interfaces (11), to provide recursively virtualizable interfaces (12), and to use activation-based communication (2).

4.1 Primordial Objects

The Viengoos virtual machine augments the hardware interface with seven objects: folios, data pages, capability pages, threads, activities, message buffers and end points. In our prototype implementation, only the first five are distinguished: a thread object encompasses a message buffer, and end points are unimplemented.

Folios A folio holds the meta-data for 128 objects. The meta-data was separated from the object to ensure that an object's size remains an easy to manage power-of-two. The meta-data 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. This includes capability slots that designate the address-space root, and current activity. It also includes an architecture-specific 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. It is orthogonal to an execution context. The intent is that an activity be used to account all basic resources including storage, memory, I/O and CPU time. Our implementation only accounts storage and memory.

An activity is able to control the resources to which its children have access. This is done by setting each child's scheduling parameters. This requires a strong reference to the activity. 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 a so-called child-relative priority and weight. This can be set via a weak-capability. This does not allow for any interference as the child-relative parameters do not affect how much memory the activity is entitled to but how to distribute the memory between an activity and its children.

When an activity is destroyed, all resources allocated to it are revoked. As any children 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.

Message Buffers A message buffer encapsulates a message includes capabilities and untyped data, and a capability slot referencing a thread to activate. Our prototype does not include a message buffer implementation. Instead, a thread object includes this functionality.

End Points An end point indirects access to message buffers. This is useful for multi-threaded servers. Our prototype does not include an end point implementation.

4.2 Resource Allocation

Whereas storage is allocated and deallocated explicitly, memory is 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 page fault, this is the activity designated by the capability in the faulting 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 activity 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.

We do not consider the costs of reading data from and writing data to backing store. This is future work.

4.3 Accounting Memory

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 claims memory that it causes to be allocated, e.g., when it accesses an object, which is on backing store.

A frame is only claimed by a single activity at a time. If a frame can be claimed by multiple activities, then a potentially unbounded amount of meta-data may be required to manage the frame. Although it is possible to attribute this meta-data to the right activity, it is difficult to ensure that the kernel can quickly recover the memory. For instance, paging such data is complicated, and arbitrarily severing claims results in less accurate accounting and could be abused. 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.

Transferring a memory claim every time an activity accesses a frame that it does not claim is too expensive: this approach causes any shared frame to ping-pong between claimants, and requires that it remain unmapped permanently so as to detect new users (it would not suffice to keep the frame unmapped for any but the thread that claimed it as a thread may execute code on behalf of multiple activities). Instead, 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*. The next activity to access such a frame claims it. In this way, the cost of shared memory is divided according to access frequency.

This accounting mechanism will present a problem when multiple activities have the same access patterns but where one always lags just slightly behind the other. In such a case, the former will tend to claim most of the memory. However, we do not consider this scenario to be realistic. Over the long term, we expect a fair distribution.

Currently, shared frames are marked as floating every two seconds. This value was chosen arbitrarily. A higher frequency would ensure a better cost distribution but also result in an increased overhead due to the added soft faults.

Periodically marking shared frames as floating can allow malicious programs to free ride by timing access to shared resources near the end of a period, that is, after it has likely been claimed. This is a problem as memory is often shared among mutually suspicious programs, primarily in the form of program and library text. To mitigate such an attack, frames should be marked shared stochastically.

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. If we were not to transfer the claim to the memory to the new user, the memory would again become active (as it was referenced) and thus be less likely to be freed, however, the original user would continue to pay to maintain the memory even though it may not again use it. By transferring the claim, the current user of the memory assumes the cost of maintaining the memory.

Viengoos uses a simple page ager to track which frames are active and which are inactive. The ager runs at 4 Hz. 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 (7). 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 more appropriate metric is required.

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 $7/8$ (88%) 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 $13/16$ (80%) 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 (19).

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 based on user-assigned priorities and the time since the last access. This process is repeated until enough frames have been removed from the in-use pool such that the in-use pool does not exceed 80% 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 activity objects: an activity object is only evicted if it has no claims and no child activity is in memory.

5. Experimental Evaluation

To test the effectiveness of our framework, we modified the Boehm garbage collector to only perform collections when the amount of allocated memory approaches or exceeds the amount of available memory.

Our test system had an AMD Duron running at 1.2 GHz with 64 KB L2 and 512 MB of RAM. We reserved 20% of the machine's RAM for Pistachio. For the tests on GNU/Linux, we used Debian 4.0.

5.1 Collector

For our experiments, we used version 7.0 of the Boehm collector (5). By default, a collection is scheduled 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 $15/16$s the available memory, 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 $7/8$s the available memory. If the heap size still exceeds $15/16$s the available memory, and if the amount of memory allocated since the last garbage collection is at least 1% of the available memory, we trigger a garbage collection and again try to discard unused memory such that the heap uses less than $7/8$s the available memory.

For our modifications to be effective, we also changed two functions to avoid reallocating discarded memory when there is still non-discarded memory available. Although the library does support unmapping unused memory, this feature is not enabled by default and thus appears to be bit rotted.

5.2 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 each tree, it immediately overwrites the pointer to the root node thereby making it available for collection.

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

		GC Time		Time	
	# GCs	Sec.	%	Sec.	Rel.
Vieng., Adapt.	108	30.8	10.9%	282.4	1.29
Vieng., Adapt., Hog	165	41.0	13.5%	301.5	1.37
Vieng., Def.	9183	232.8	52.5%	443.2	2.02
Vieng., Def., Hog	9189	255.2	53.6%	475.7	2.17
Linux, Fixed	98	14.4	6.5%	218.9	1
Linux, Fixed, Hog	98	30.6	9.4%	325.5	1.48
Linux, Def.	9325	212.8	53.1%	400.4	1.82
Linux, Def., Hog	9294	213.9	52.9%	403.7	1.84

Table 1. The number of collections, the time spent collecting, and the time to execute for each configuration.

We modified this benchmark in two ways. First, we loop over the tree creation 100 times, which lengthens the execution time and allows us to better measure the effectiveness of the scheduler. Second, we introduced a memory hog. When enabled, shortly after the benchmark starts, it allocates 5 MB of memory per second until it has allocated half the initially available memory (in our case, approximately 170 MB). After sleeping for a minute, it then deallocates the memory, again at a rate of 5 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 350 MB of memory. This is approximately the amount of memory available to the benchmark on Viengoos when there is no memory hog.

When running the benchmark under GNU/Linux, we first allocate and lock 100 MB of memory. This represents the approximate difference in the amount of memory available to user-land programs on our system running GNU/Linux and on our system running Viengoos. The main issue is the amount of memory that we statically reserve for Pistachio. Note that without the memory hog, the time to complete the benchmark is the same for the case where we lock the memory as for the case where we do not.

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.

5.3 Results

Table 1 shows the number of garbage collections, the time spent in the garbage collector, and the execution time for each of the eight configurations. Figure 4 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. Garbage collection pause times can be recognized by the regular short ledges. Figure 5 shows a plot of time vs. the benchmark's heap size. This shows the influence of the memory hog on the benchmark's memory use and execution time.

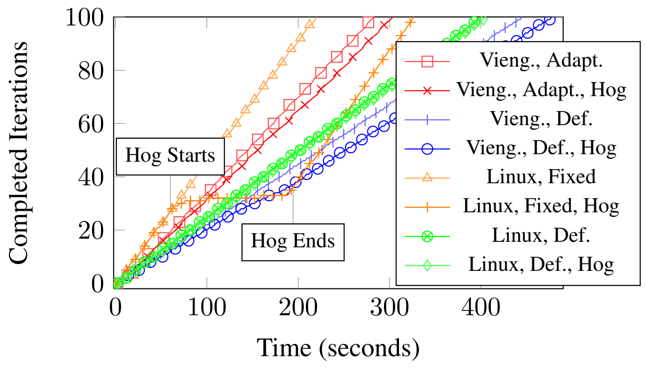

Figure 4. Time vs. iterations for Viengoos and GNU/Linux, using either the adaptive, fixed or default scheduler, and with or without a memory hog.

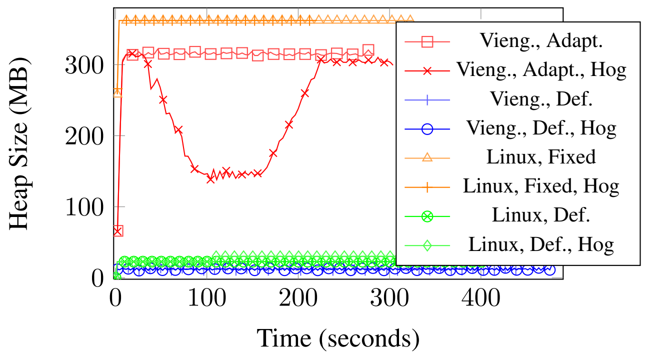

Figure 5. Time vs. heap size for Viengoos and GNU/Linux, using either the adaptive, fixed or default scheduler, and with or without a memory hog.

We first note that the modified scheduler significantly reduces the time spent in the collector. On both systems, when using the default scheduler, the benchmark spends more than 50% of its time in the collector. This is reduced to less than 15% when using one of our schedulers. We do not expect this speed-up to be typical: the benchmark specifically exercises the collector and does little actual work.

Second, on Viengoos, the presence of the memory hog results in only a marginal slowdown. This is both the case for the default scheduler and the adaptive scheduler. The adaptive scheduler is able to adapt and simply use less memory. This is shown in figure 6: as the memory hog allocates memory, the amount of memory reported to the garbage collected program decreases accordingly. When the memory hog again frees the memory, the amount of available memory increases. This is reflected in the reported availability. The garbage collector immediately adapts to exploit this memory.

When the memory hog has reached its pinnacle, it appears that the system reports that there is more memory available

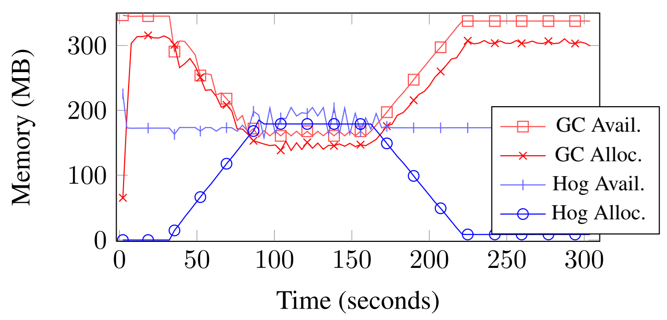

Figure 6. Time vs. memory, either allocated or available, for the benchmark and the memory hog running on Viengoos, using the adaptive scheduler.

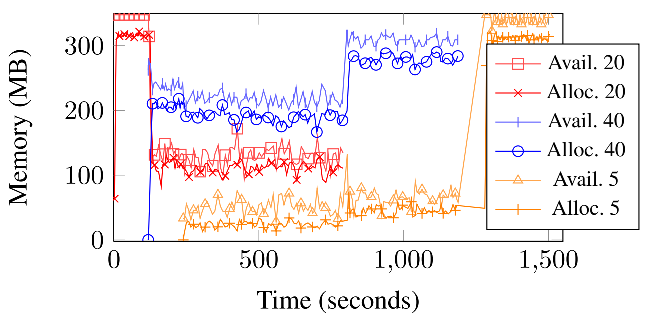

Figure 7. Three garbage-collected applications.

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 available memory reported to the garbage collector.

The execution of the benchmark in the presence of the memory hog with the default scheduler is not noticeably impacted on either system simply because the default scheduler significantly restricts the amount of memory that it uses.

On GNU/Linux with the memory hog, the fixed scheduler performs, as expected, poorly: while the memory hog is active, the garbage collector thrashes and makes no noticeable progress. It is only because the memory hog stops that the benchmark even completes in a reasonable time. This is easily seen in figure 4: while the memory hog has allocated its share, the benchmark makes no progress.

To determine why the benchmark runs slower on Viengoos than on GNU/Linux, we profiled our kernel and application. We identified three major slowdowns. The first is the use of a page ager. Viengoos collects reference bits using Pistachio's `l4_unmap` function. Although we use batching, *`l4_unmap` represents 13% of the execution time*. The next is due to address-space management, which took about 10% of the execution time: although `mmap` accepts ranges, on Viengoos, each page is managed individually in user space. This overhead could be reduced without violating the object model by way of an iterator object, which invokes the same method on an array of capabilities. Finally, page faults take 5% of the execution time. Using L4, faults are reflected twice: once when accessing an invalid region, and again, after the application has installed a page object but before Viengoos has established an L4 mapping.

5.4 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 20, 40 and 5. Their starts were staggered.

Figure 7 shows a plot of time vs. the amount of memory available or allocated to each activity. When the system starts, the first activity immediately uses all the memory. When the second instance starts, it allocates its share and the first instance adapts. When the third instance starts, the first two instances yield enough memory such that the new instance is allocated its share. As instances complete, the remaining instances adapt to use the newly available memory.

6. Related Work

Significant work has been done exploring how to allow applications to better manage the resources available to them. V++ (16), exokernel (10), Nemesis (14) 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" (10).

On V++, memory is distributed using a market-based approach (15). 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 tries to avoid imposing policy. To this end, it uses a distributed approach to managing memory: physical resources are distributed and visible revocation is used to revoke them. The SawMill authors do not address how to distribute resources among multiple principals. This necessarily imposes some policy. If this architecture were used for a general-purpose system, some *de facto* policy would develop simply because multiple applications must negotiate and the choice of language is policy. Finally, 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.

Iyer observes that fully-transparent virtual memory has become a serious impediment to "performance-minded programmers" (18). Iyer's solution is that the memory manager maintain and provide interested applications access to a so-called severity metric that describes memory pressure as well as the cost of paging. Using this information, cooperative applications can continuously and proactively adapt their memory consumption to available memory. Relative priorities are expressed using nice values, which are used to scale the severity metric. This makes using the severity metric to determine whether to free or to page memory useless.

Yang et al. observe that to maximize a garbage collector's performance, it should use the largest possible heap size that fits in memory, however, current systems do not provide the necessary information (24). To facilitate such adaptations, they develop a heap-sizing model, which works by reporting the current working-set size and a target working-set size to interested processes. A garbage-collected program, however, does not conform to this model: during normal execution, it has a small working set relative to its heap size; during a collection, it references all live objects. As we note, using working sets to determine demand is ineffective.

7. Concluding Remarks

We argued that many programs have the ability to adapt, and that doing so is becoming more important given the increasing range of hardware configurations on which software is expected to run. We observed that adapting is not only problematic because general-purpose operating systems do not provide a mechanism to determine how resource availability, but that the current resource distribution scheme is inappropriate for adaptive applications, which cause a positive-feedback loop and ineffective scheduling.

We described a resource management framework that incorporates stakeholder preferences and converges to a configuration with maximum expected utility by way of a negative-feedback loop. We presented an algorithm to compute the amount of available memory for each process, and show how to account shared memory. Our experimental results show that our framework correctly handles multiple adaptive applications simultaneously.

References

[1] R. Alonso and A. W. Appel. An advisor for flexible working sets. In *Proc. of the 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 *Proc. of the 13th 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 Comp. Syst. Arch. Conf.*, Jan. 2001.

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

[6] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *VIS '97: Proceedings of the 8th conference on Visualization*, 1997.

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

[8] P. J. Denning. *In the Beginning: Recollections of Software Pioneers*, chapter Before Memory was Virtual, pages 250–271. IEEE Computer Society Press, 1997.

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

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

[11] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the fluke kernel. In *Proc. of the 3rd OSDI*, pages 101–115, 1999.

[12] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. *Proc. of the 2nd OSDI*, Oct. 1996.

[13] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. Technical report, University of Texas at Austin, Austin, TX, USA, 1996.

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

[15] 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.

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

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

[18] S. Iyer. *Advanced memory management and disk scheduling techniques for general-purpose operating systems*. PhD thesis, Rice University, Houston, Texas, November 2005.

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

[20] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

[21] J. H. Saltzer. Naming and binding of objects. In *Operating Systems, An Advanced Course*, pages 99–208, 1978.

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

[23] System Architecture Group. The L4Ka::Pistachio microkernel. Technical report, University of Karlsruhe, May 2003.

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