30 Years of Memory Mismanagement is Enough! Improving the Memory Residency Problem

Neal H. Walfield Jonathan S. Shapiro
Johns Hopkins University
{*neal,shap*}*@cs.jhu.edu*

Abstract

Many programs could improve their performance by adapting their memory use. To ensure that an adaptation increases utility, a program needs to know not only how much memory is available but how much it should use at any given time. This is difficult on commodity operating systems, which provide little information to inform these decisions. As evidence of the importance of adaptations to program performance, many programs currently adapt using *ad hoc* heuristics to control their behavior.

Supporting adaptive applications has become pressing: the range of hardware that applications are expected to run on—from smart phones and netbooks to high-end desktops and servers—is increasing as is the dynamic nature of workloads stemming from server consolidation. The practical result is that the *ad hoc* heuristics are less effective as assumptions about the environment are less reliable, and as such memory is more frequently under- or over- utilized. Failing to adapt limits the degree of possible consolidation. We contend that in order for programs to make the best of available resources, research needs to be conducted into how the operating system can better support aggressive adaptations.

1 Introduction

Programs can often adapt behavior in response to available memory. For instance, the time that a garbage collector spends in the collector can be reduced by using a larger heap [1], and computed results can be cached and reused reducing latency and increasing throughput [8]. Adaptations are not required for the correct execution of a program; they improve the program's utility—assuming, that is, that the extra memory could not have been better employed elsewhere.

There are two challenges in intelligent adaptation. First, a programmer must determine whether the resources required for the adaptation are available. Using too much memory will result in paging, which often negates any benefits an adaptation may have brought. Second, to maximize the system's utility, a program needs to coordinate its resource use with that of other adaptive programs in the system. Commodity operating systems do not enable either of these requirements.

The lack of support for adaptive applications stems from the use of transparent multiplexing. Transparently multiplexing resources simplifies programming by removing the need for programmers to consider how to schedule available resources by way of complex overlay strategies, and thus allows them to concentrate on solving the functional aspect of their problem [3]. The result, however, is that programs have no reliable mechanism to determine what resources are available to them. Without this knowledge, any adaptation becomes a gamble. This gamble is a costly one given that a bad adaptation results in unnecessary paging or even thrashing [14]. Further, because transparent multiplexing uses program behavior to estimate demand, programs that manage their own resources cause the scheduler to reach the wrong assumptions, and may even create a positive feedback loop as the program tries to determine the appropriate allocation and the system readjusts to perceived demand.

In this paper, we sketch possibilities for how to allow programmers to provide high quality results by exploiting available resources while remaining neighborly. The underlying idea is to introduce mechanisms that allow unpremeditated coordination of resources among the running programs such that the system constantly converges towards a configuration with maximize expected utility.

2 Today: *Ad Hoc* Adaptations

Many garbage collectors can reduce the time they spend collecting by using a larger heap. This is because the time it takes to perform a collection is proportional to the size of the live objects, not the heap size [1]. Allow-

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 using ImageMagick v6.3.5.10's `convert`. Output was in RGB raw format and sent to `/dev/null`. Scaling was done using the `-geometry` option.

ing the heap to grow too large is problematic as paging or thrashing can result [14]. This often offsets any savings. Ideally, the garbage collector should perform a collection just before paging is initiated. As the operating system provides no feedback, this is difficult to time. In practice, garbage collectors are space conservative and use small heaps. The Boehm collector, for instance, performs a collection when approximately 50% of the size of the heap after the last collection has been allocated. This amortizes garbage collection overhead but does not exploit extra memory.

Another useful adaptation is caching. 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. According to Nielsen [10], 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. Ideally, the cache should grow to fill the available memory and the application should be able to shrink it when there is pressure.

Caching is also effective in server programs. For instance, the Deens DNS server, was converted to cache queries and the result was a near doubling of the program's throughput [8]. Many server programs also maintain a pool of worker threads to more quickly dispatch requests. This should be sized in the same way. The slab allocator maintains a cache of initialized but unused objects, which it could allow to grow to the amount of available memory and shrink in reaction to memory pressure.

Reliably determining how much memory is available and freeing memory when there is pressure is difficult. Given the benefits of using extra memory, many programs employ *ad hoc* techniques to determine how and how much to memory to use. A common policy used to size a cache is for the programmer to select a cache size *a priori*. A more dynamic policy sizes the cache according to use by freeing entries that have have not been accessed for some amount of time [12, 11]. Both of these policies are fragile because neither considers the most important factor: the amount of available memory.

Today, the range of execution hardware configurations is wider and less well calibrated. In 2008, smart phones and netbooks designed to run general applications became popular, and Citrix announced a virtualization environment to run Windows applications on the iPhone [6]. Industry wants to save money by reusing existing components [9], and generate revenue by having existing components reach more users. Meanwhile, users want to run the same applications everywhere [6]. To ensure good performance on these resource-poor machines, programs need to update their assumptions. They cannot, for instance, use non-adaptive caches [7]. For these programs to remain competitive, they need to also take advantage of the available resources on larger machines.

We also suspect that *ad hoc* sizing techniques will become an increasing problem for servers. Server consolidation continues to grow in popularity and save money on both hardware and power. The effect is that server programs are running in more dynamic environments. For such configurations, static adaptive policies—if they work at all [14]—are unlikely to provide as good degrees of consolidation as policies informed by the current system status. Although ballooning helps make virtualized operating systems more adaptive [13], it does not go far enough in that the adaptations that we are interested in require that the applications collaborate. For these types of adaptations, new mechanisms are required.

The lack of support for the type of adaptations we have discussed is fundamentally different from the type of adaptations that the exokernel design tries to enable. In exokernel, the main design goal is to avoid abstractions as much as feasible by providing user-programs with as low-level access to resources as possible [5]. Programs running on an exokernel can determine not only what physical resources are assigned to them but can exploit these resources' low-level capabilities. This solution does not address the overarching issue of how to allocate the system's resources. This suggests that the policy decentralization advocates are wrong: effective systemic performance requires coordinated systemic policy.

Ad hoc adaptation techniques were survivable when programs ran on a narrow and well-calibrated range of hardware. This is no longer the case. For applications to effectively exploit their environment, they need to adapt and this requires operating system support.

3 Supporting Adaptive Applications

Two issues hinder effective adaptations. First, the working set model of program behavior interprets utility as a

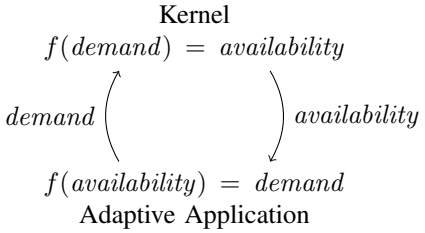

Figure 1: Adaptive applications create a positive-feedback loop when managed according to working set principles. Although the system may stabilize, no mechanism ensures that it converges to a configuration with maximum utility.

function of recency of use. This does not match how adaptive programs use memory. As such, new memory allocation schemes should be investigated. Second, in addition to providing the information programs need to adapt intelligently and facilitating collaboration, operating systems can increase the precision and agility of adaptations by actively participating in the adaptation process by performing adaptations on applications' behalves. In this way, adaptations are only used if necessary and only as much as necessary. The challenge is to maintain robustness in the face of potentially increased complexity and misbehaving programs.

3.1 Memory Allocation

Enabling effective adaptation on general-purpose operating systems is not just a question of supporting programs in determining how much memory is available: general-purpose operating systems allocate resources based on observed demand, however, demand is exactly what adaptive applications vary according to availability. This is illustrated in figure 1. This can result in a positive-feedback loop, and, ultimately, ineffective scheduling. An application perceives a certain amount of available resources and adapts by changing its demand. The operating system then sees a change in demand and rebalances the current allocation. The application again detects a change in availability and again adapts its use, etc. Although this may eventually stabilize, there is no reason for the resulting allocation to reflect a configuration with high utility.

The purpose of adapting is to increase a program's utility (that is, its worth to its stakeholders). We argue that a scheme should be explored in which the system distributes memory according to stakeholder-specified expected utility.

The allocation and availability information should also be temporally quantized. For instance, if a program sees that there are several gigabytes of memory available to it, and commits to a computation that uses that memory for several seconds but then has the memory revoked just milliseconds later, the adaptation is ineffective and may result in negative utility.

3.2 Precise and Agile Adaptations

Many useful adaptations such as determining how to size a cache and how to size a heap can by realized just by knowing how much memory should be used in the near future. Assuming that this information is made available or somehow negotiated, the process needs to monitor the availability and alter its behavior accordingly.

There are three shortcomings with this approach. First, if a process does not adapt quickly, it may be viewed as uncooperative and preemptively paged. As paging is often more expensive than simply using a smaller cache, avoiding the risk of being forcefully paged is an incentive to not fully exploit all the available resources. Second, when demand changes because some process begins to use more memory, some processes will observe that less memory is available and adapt. This may create a herding effect in which more processes adapt than is necessary. Finally, if a process adapts to every small change in system state, it may spend more time adapting than doing useful work.

These issues can be avoided by having the resource manager perform the adaptations on behalf of the programs. Because the resource manager would in this case be the actor, adaptations can be executed at the very last instant and only to the degree necessary to relieve pressure. The challenge with this approach is that the mechanisms need to be designed carefully so that they do not overly complicate the memory manager or require it to depend on untrusted processes, both of which would decrease its robustness [4].

4 Memory Management in Viengoos

We have begun to explore how to improve memory management in the context of an experimental operating system called Viengoos. Viengoos is a clean slate design with the goal of providing a platform for experimenting with novel resource management schemes and interfaces.

Work is currently being conducted on two fronts. First, we are examining how to coordinate access to physical memory so as to maximize expected utility. Second, we are considering how to increase the precision and agility of adaptations by studying common types of adaptations and identifying simple and salient information that a program can communicate to a central resource manager, which can then safely perform the adaptations on the program's behalf.

		GC Time		Time	
	GCs	Sec.	Percent	Sec.	Relative
Adaptive	108	30.8	10.9%	282.4	1
Default	9183	232.8	52.5%	443.2	1.56

Table 2: Number of garbage collections, time spent collecting, and time required to complete the benchmark using the adaptive scheduler and the default scheduler.

4.1 Memory Allocation

Based on the observation that the parent is usually the primary stakeholder, Viengoos distributes resources according to a hierarchical model. A parent process assigns a priority and a weight to each of its children. These values are used by the resource manager to determine each process's access to its parent's memory allocation in a highest-priority first, proportional share manner. These parameters were selected because they are simple and appear to succinctly capture stakeholder preferences, which is exactly what a utility function does.

When determining how to distribute memory, Viengoos maximally assigns a process as much as it currently demands. In this way, the scheduler is work conserving. This allows lower-priority applications to make use of otherwise idle resources.

Using the same parameters, we have also developed an algorithm to compute the amount of memory that is approximately available to a process. This allows a process to straightforwardly determine how much memory it could use in the near future. Both a process's current allocation and availability are recomputed and published on a regular basis.

To demonstrate the effectiveness of this approach we modified the Boehm garbage collector to size its heap according to the amount of reported available memory. Normally, the Boehm collector tracks the number of bytes allocated since the last collection and when this exceeds approximately 50% of the heap size after the last collection, it performs a collection. This limits the amount of garbage to a constant factory larger than the live objects at the last collection. This strategy does not take advantage of additional memory to delay collections. Because collection time is proportional to the number of live objects and not heap size [1], this can be significant [14]. Our benchmark allocates some data structures, links them, overwrites the root pointer (thereby making the data structures garbage), and then loops. It was run on an AMD Duron running at 1.2 GHz with 64 KB L2 and 512 MB of RAM. The results of running the benchmark with the adaptive scheduler and the default scheduler are summarized in table 2. The adaptive scheduler reduced the total number of garbage collections by almost two orders of magnitude and the total execution time by more than a third.

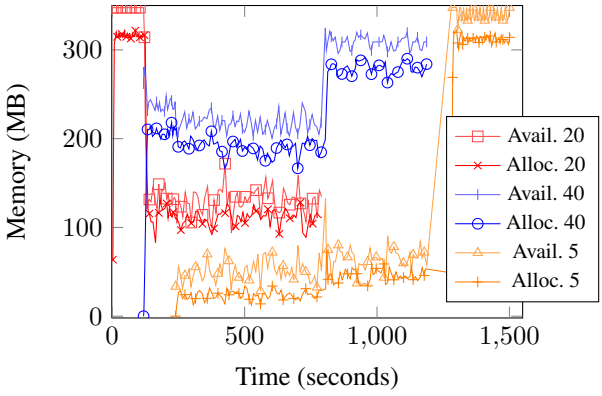

Figure 2: Three instances of a garbage-collected program with a common parent and different assigned weights adapting their heap according to the amount of reported available memory. The program instances' starts are staggered. The graph shows the time since the start of the benchmark vs. the memory available to an instance and the amount of memory that an instance uses.

The real test is how well this works in the face of multiple adaptive applications. To test this, we ran three instances of the above benchmark with a common parent. Each was assigned the same priority but a different weight. The starts were staggered to show the adaptation. Figure 2 shows a plot of time vs. the amount of memory available or allocated to each instance. When the benchmark starts, the first instance 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 last instance is allocated its share. As each instance completes, the remaining instances adapt to use the newly available memory.

4.2 Increasing Agility and Precision

A well-studied example of how to provide simple and safe management information to the kernel is page rankings [2]. This information is simple to express and captures a useful memory management policy. Further, it can be used in such a way that either the information only hurts the application itself or it represents a non-negative improvement in system performance.

We have identified another example of this type of information: discardable memory. We observe that it is often cheaper to discard memory and recompute when it is next required than to save it to disk when its physical memory is preempted. To support discardable memory, the resource manager needs to provide an interface to allow programs to indicate that memory is discardable.

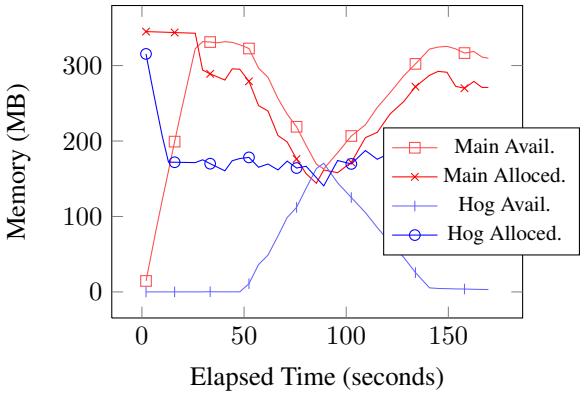

Figure 3: Time vs. memory (available or allocated) for a caching program whose cache is discardable (and thus managed by the resource manager) and a memory hog.

The manager uses this information when it selects a page for eviction. If the page has been marked as discardable, it can reuse the frame without saving the contents. However, the resource manager must note that that page has been discarded and signal the application if it tries to use it again. In Viengoos, we signal this information at the time of the next access. This ensures that any user of the page receives the signal, and, when the signal is sent, that the recipient is necessarily ready to receive it (the accessing process just faulted).

To demonstrate the effectiveness of discardable memory in realizing adaptations, we wrote a program that manages a cache of objects each one megabyte large and which accesses the objects according to a Zipf distribution. If an object is not in the cache, the program generates it by querying a database (managed by SQLite) and marks the object as being discardable. The program then accesses the object. If it has been discarded, the program is signaled. It calls the object creation mechanism and resumes execution. This requires a dozen lines of code.

When the program exceeds the memory available to it, the resource manager automatically finds the appropriate pages using, e.g., a LRU policy and discards them. This is the desired adaptation. For finer control, this could be augmented using page rankings to ensure that pages in the cache are selected before other program data.

We also introduced a memory hog, which after some time slowly allocates half the memory in the system, holds onto the memory for a while and eventually slowly releases it. As the memory hog allocates memory, the resource manager automatically shrinks the main program's cache. The execution of the benchmark in the presence of the memory hog is shown in figure 3. As can be seen, the resource manager automatically shrinks the benchmark's cache as required.

5 Conclusion

Applications are expected to run on an increasingly wide range of hardware configurations and in more dynamic environments. Our initial experiments suggest that operating system support for adaptation can reduce global memory requirements, improve systemic performance, or both. Adaptation requires a reexamination of operating system support for adaptive applications, which has traditionally been ignored by commodity systems forcing application to use *ad hoc* adaptation strategies. These strategies are no longer reliable.

Enabling effective adaptations requires new memory management schemes that explicitly consider adaptive applications, and operating system support of applications to maximize adaptation precision and agility. We have presented Viengoos, a system that we are using to explore these problems.

References

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

[2] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *USENIX Summer 1994 Technical Conference*, June 1994.

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

[4] P. Druschel, V. S. Pai, and W. Zwaenepoel. Extensible kernels are leading OS research astray. *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, May 1997.

[5] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Dec. 1995.

[6] 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.

[7] J. Gettys. $100 laptop / OLPC (One Laptop Per Child). `http://gettysfamily.org/wordpress/?p=11`, Nov. 2005.

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

[9] R. Needleman. Technology marches backward. `http://reviews.cnet.com/4520-3000_7-6542073.html`, June 2006.

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

[11] S. Parmenter. Firefox 3 memory usage. `http://blog.pavlov.net/2008/03/11/firefox-3-memory-usage/`, Mar. 2008.

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

[13] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[14] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006.