

Application-assisted physical memory management

Sitaram Iyer

Juan Navarro

Peter Druschel

{ssiyer, jnavarro, druschel}@cs.rice.edu

Rice University, 6100 Main Street, MS-132, Houston, TX 77005, USA.

Abstract

Many software applications can, in principle, trade main memory consumption for other resources. For instance, garbage collected language runtimes can trade collection overhead for heap size, and many programs can improve their performance by caching data that was precomputed, read from disk or received from the network. Unfortunately, OSs provide little useful information about physical memory availability, and so applications use main memory cautiously to avoid unnecessary paging when memory is scarce.

In this paper, we revisit physical memory management in general-purpose OSs, with the goal to inform interested applications about the level of contention for physical memory. We define a severity metric for memory contention, which allows applications to balance the cost of paging against the cost of freeing and regenerating a memory page. We describe how the system can maintain the severity metric with low overhead, and argue that slightly modified applications can realize substantial performance improvements by dynamically adjusting their memory consumption to physical memory availability.

1 Introduction

In the early days of computing, the limited amount of physical memory posed a hard limit on the size of programs. With the invention of virtual memory, physical memory management got hidden inside the OS, and applications were presented a large address space backed by physical memory and secondary storage. In practice, however, applications face a dramatic performance penalty if they collectively overflow the system's available physical memory. They are often run in environments unforeseen at development time, and are subjected to unpredictable workloads. Applications are therefore compelled to be unnecessarily conservative about their memory use, even though they could often improve their performance by using more memory when it is available.

On the other hand, in spite of programmers exercising such caution, an unexpected burst of load can still exhaust physical memory and suddenly induce heavy paging activity and degrade

performance to unacceptable levels. This paging may happen partly for memory that could easily have been freed by the application, if it were involved in physical memory management. We contend, therefore, that the virtual memory abstraction renders applications oblivious to physical memory availability, including applications that could use such information to improve their performance and that of the overall system. The following paragraph exemplifies a variety of ways in which this is possible.

Many modern applications use algorithms that are *elastic* in terms of their ability to trade memory consumption for performance or the usage of some other resource. For example, language runtimes such as Java virtual machines can release unused memory through garbage collection. Many malloc implementations tend to retain the memory that applications free, thus reducing the number of system calls they issue; this memory can potentially be released to the system when there is a shortage of free memory. Applications such as databases and web browsers that maintain a memory cache of disk or network data could increase performance by enlarging the cache, or alleviate memory pressure by releasing some of this memory. Many algorithms can trade CPU time for memory by retaining different amounts of partial results that may be reused at later stages in the computation.

This paper proposes a memory management system that takes advantage of elasticity in applications to improve performance, robustness and user control over memory allocations. It is based on the following four ideas:

(1a) The system should *notify* applications about impending memory pressure, and give applications a chance to *react* to it. Being aware of this, elastic applications can safely allocate large amounts of memory; when notified, they can release memory by performing actions that are less costly than paging. These applications may restore their memory allocations if free memory becomes available again, thus operating the system close to full memory utilization.

(1b) Furthermore, the system should expose information about the *severity of memory pressure*, using a suitably chosen metric. This enables applications to compare the cost of performing a certain memory releasing action against the I/O costs being incurred for paging, so that they can gracefully respond

to increasing memory pressure by taking increasingly drastic actions.

These two ideas – notifications along with severity information – can increase the performance of elastic applications when excess memory is available, as well as improve robustness in memory constrained circumstances.

(2a) A system may be running several elastic applications, each of which may autonomously choose among several actions to reduce memory consumption when notified by the system. It is therefore desirable to impose a usable scheme of *preferences*, ultimately under user control, to determine which applications should first take actions and to what degree. For example, interactive applications can be asked to be less aggressive than background processes about shrinking their caches.

(2b) Finally, a natural and useful extension of this preferences model would be to bias the page replacement policy with these preferences. This would, for instance, permit memory from interactive or soft real-time applications to be paged out only when memory pressure is severe, whereas background processes can become attractive candidates for page eviction.

The key idea that connects these four concepts is a novel metric to quantify the severity of memory pressure, defined based on the recency of the last access to the page that is the next candidate for replacement. This *severity metric* enables us to establish a simple quantitative relationship between the memory contention in the system, the type and degree of applications' reactions, and the preferences model.

2 Severity metric based on page age

This section proposes a metric for reporting the severity of memory pressure to applications. We postulate the following four requirements.

(1) The metric should characterize memory pressure, especially the following two aspects: the cost of page replacement on application performance when free memory is unavailable, and the cost of imminent paging when free memory is available but close to exhaustion.

(2) The metric should provide a common and clearly defined reference point for applications to quantify and compare the impact of their actions against, and independently decide whether to perform an action at a given time.

(3) The metric should lend itself to some intuitive scheme for user specified preferences, by weighting the value of the metric.

(4) The metric should be cheap to compute and maintain.

Now we propose a metric that fulfils all four requirements, show how it can be used by applications, and describe how preferences can be implemented.

Definition of the severity metric: We define the *page age* of a physical memory page to be the time (in seconds) since

it was last referenced. When the system has exhausted its free memory, we define the system-wide *severity metric* for memory pressure to be the age of the page that is the current candidate for page replacement. Note that a smaller value of the metric depicts a higher level of memory contention. The system may bound the metric by reasonable limits, say one second (S_{min}) and one hour (S_{max}).

When the system is paging, it informs applications of the current severity metric. Applications need to compare the paging cost against the cost of regenerating some memory that they can potentially free, so as to decide which alternative is cheaper. They can use the LRU principle to assume that the page that is being paged out may not be accessed again for approximately as much time as its age. This enables them to calculate the *amortized paging I/O cost* by dividing the disk service time by the expected time to next access, i.e., the page age. They can compare this paging cost against the cost incurred if the application memory is released (i.e., the time overhead of freeing the memory, and subsequently recomputing or refetching its contents when they are required), amortized over its own expected time to next access (e.g., the age of this memory). Thus, the metric enables paging to serve as a common reference point to compare against application-specific costs of freeing memory. Furthermore, this achieves a global *least-impact-first memory replacement*, where the physical page or the application buffer that has the least overall impact on performance is replaced first.

Secondly, if the amount of free memory is small, i.e., between zero and a system-defined threshold (say 10% of system memory), then the severity metric is gradually decreased from its maximum value to its required value. It is computed by geometrically interpolating between S_{max} (signifying no memory pressure) and the target severity value (i.e., the age of the current page replacement candidate if paging were to occur). The rationale is that if free memory drops linearly from 10% to zero, then the reported severity metric will drop *exponentially* from S_{max} to its target value. This conveys the urgency of imminent paging only when free memory is low enough, thereby balancing the needs of early notifications against the desire to operate the system close to full memory usage. Also, it allows applications some time to take actions, especially time-consuming ones like garbage collection.

Finally if free memory is available in plenty, so that paging is not likely in the near future, then the severity metric is set to S_{max} .

Use of the severity metric: The system periodically recomputes the metric every few seconds, and makes it available to applications through a new system call named *severity*. Each time the metric significantly changes, such as when it doubles or halves, a new signal called SIGMEM is used to notify all applications. Applications are free to ignore this signal; however, interested applications may implement signal handlers that query the severity metric and then potentially decide to release some

memory. These signal handlers may reside in different layers of software, such as in application code, libraries or language runtimes: in the latter cases, the main application may not need to be modified. Finally, there is no need for a special API to express user-specified preferences: we simply extend Unix-style per-process CPU `nice` values to specify *weights* for the memory subsystem.

The following are four types of increasingly drastic application actions, where the application may react in the described manner to the severity metric notification, and achieve the least-impact-first replacement described earlier.

(1) When the free page list is short and shrinking, the reported severity metric decreases exponentially towards its target value, and triggers inexpensive memory releasing actions that are taken before paging even starts. For example, garbage collection and releasing malloc-held unused memory are typically cheaper than paging.

(2) If the above actions are unsuccessful in releasing enough memory to avoid paging altogether, then free memory exhausts and paging starts, and the severity metric progressively decreases. Some applications (such as databases) maintain caches of file data, so that the regeneration cost is equal to I/O cost. They can release buffers that were accessed less recently than the severity metric, so that they prevent the system from unnecessarily paging out file contents to swap space, and also extends global LRU replacement to application caches.

(3) Some applications maintain buffers (such as database query caches or web caches) whose regeneration may involve several disk requests or may require the use of some other resource such as CPU or network. As described earlier, these applications need to empirically estimate the regeneration costs and express it in time units.

(4) If the system does start thrashing despite all the above, then it may become too unresponsive to remain useful. Applications may then choose to react to very low values of the severity metric by disabling some memory intensive features, or switching to alternative algorithms that use less memory, thereby sacrificing functionality or consistency, or increasing the consumption of some other resource, but restoring stability in the system.

We assume that applications cooperate with the OS to improve overall system performance. Faulty or malicious applications can at worst cause excessive paging by allocating too much memory, ignoring memory pressure notifications. This is essentially no different from current general-purpose OSs. Moreover, we next describe, our system enables applications to be isolated from such rogue applications by weighting their memory page ages.

Preferences by weighting processes: In our system, it is useful to be able to discriminate between applications in terms of how aggressively these applications' memory-releasing actions are taken, and to what extent their pages are preferred for re-

placement. For example, a *system designer* can implement a policy that identifies and prefers interactive applications over others; a *programmer* for a background download application can *weight down* its own memory, making it more attractive for eviction; and a *user* can request for say an email application to be left untouched while the rest of the system may be paging.

A crude form of preferences is to enable only the complete prioritization of one process from others, so that the former is effectively pinned to memory until the other processes are fully paged out and their actions fully invoked. However, in a general-purpose OS, one would typically prefer a continuous gradation of *weights* to processes, so that drastic actions and recently accessed pages (e.g., code, static data) from processes with lower weight compete with mild actions and older pages (e.g., cache objects) from *weighted processes*.

From the previous section, the amortized paging costs for a process can be amplified by *linearly scaling down* the age of all its pages by a weight. Page replacement is performed in order of the resulting *scaled ages*, and the global severity metric is the scaled age of the current page replacement candidate. Observe that this weighting is transparent to applications, whether or not they are modified for our system.

This weighting concept can be extended from paging to application actions in a manner transparent to applications. Rather than reporting this global severity metric identically to all applications, the system reports the value of the metric scaled by the *inverse of the weight* of the notified process. Thus, a *weighted down* process is informed that the severity is relatively higher, and is induced into taking more drastic actions.

3 Design

This section describes the design of the in-kernel portion of our system, which at its core is an *age-aware page daemon*. We show how it maintains page ages and uses them as the page replacement criterion, how it efficiently weights processes, and how it achieves some interesting properties as a result.

Age-based buckets: Our page daemon classifies memory pages into *page buckets* of different age ranges. These age ranges increase exponentially in powers of 2, such as 1–2s, 2–4s, 4–8s, etc., so that a small number of buckets suffice to cover a large age range with accuracy proportional to the age. The bucket index, therefore, corresponds to the logarithm of the page age. For more fine-grained accounting of page ages, this ratio 2 can be refined to say $\sqrt{2}$.

The page daemon scans the page table entries for the pages in these buckets, as often as the smaller age range bound of the bucket. If a page in this bucket is found to have its reference bit clear, indicating that the page has not been referenced since the last scan, then the page is moved to the next bucket. Otherwise

the page has been accessed in the interim; its reference bit is cleared and the page is moved to the first bucket.

Instead of a single global list of all page buckets, the system maintains one list for each process, keeping track of virtual pages as mapped by and referenced by that process. Buckets from different processes with the same index are *chained* together on one chain per bucket index. When a page needs to be chosen for LRU page replacement, these chains are traversed in order of decreasing age (right to left in Figure 1) and a page is randomly chosen from the first chain encountered with non-empty buckets.

Weighting: Conceptually, when a process is weighted, the system needs to scale the age of the process’ pages by dividing their ages by some factor, and then reorder all pages in the system by their scaled ages for page replacement purposes. This can be made efficient by avoiding sorting and division operations, as follows. Adjacent buckets in any bucket list differ in their age range by a factor of 2. To weight a process by dividing the age of all its pages by W , we simply have to shift each of its buckets one chain to the left by $\log W$ steps (refer to Figure 1). The absolute index of the bucket after this shift is the scaled age of the pages in the bucket, and is also the *severity* of memory pressure if the page is evicted. Because weighting only involves detaching a small number of buckets per process ($B = 3$ in the figure; about 10 in practice) from their respective chains and reattaching them in different chains, it is very efficient.

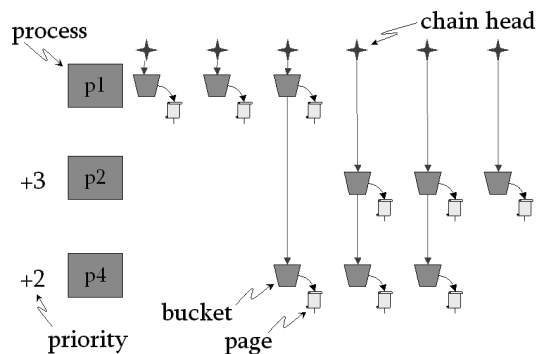


Figure 1: Buckets.

It is convenient to extend Unix-style CPU nice values (typically in the $[-20, 19]$ range) directly to the memory system. This priority can be linearly scaled onto the $[-S/2, S/2]$ weight range, where we designate S as a system-defined scaling factor (30 in our implementation) indicating how significant the impact of weighting is; it may be equal to a small multiple of the number of buckets per process B .

Priority classes and overlap: Observe in the figure that the buckets of processes p_1 and p_2 do not overlap in their index ranges, due to a combination of system-imposed limits on page

ages and sufficient weighting. These processes have been assigned to different *priority classes*, so that processes in one class cannot induce higher priority class processes to release memory, either by paging or by triggering application actions. There are S/B disjoint priority classes (3 in our implementation). In contrast, note that processes p_1 and p_3 in the figure overlap to some degree, so the recently accessed pages of p_3 compete with the older pages of p_1 , achieving the desired type of overlap.

Weighting dirty pages: Dirty pages are more expensive to evict than clean pages, since they need to be first written to disk, and then paged in if accessed later. Hence we weight the dirty pages by a factor of two, by associating them with a separate set of buckets and shifting them one position to the left. We can also exercise additional flexibility by dynamically varying the amount of dirty page weighting by the amount of disk contention.

More uses of weighting ages: Age weighting provides a versatile framework for two more techniques briefly mentioned here. The page replacement policy can be biased by frequency of page accesses, akin to activation counts in Linux and FreeBSD. It is also possible to provide a novel facility of *partial memory isolation*, which converts the two existing extremes (total isolation [5] and no isolation) into a continuum.

4 Preliminary evaluation

We implemented a prototype of the system in the FreeBSD-4.3 kernel and ran it on our desktop workstations. The behaviour is as expected: when paging starts, pages from dormant background processes with ages of $S_{max} = 1$ hour get evicted. Subject to moderate load with sporadic paging, the severity metric drops to a few minutes or even to seconds. Rapid and extensive memory access causes the metric to fall to $S_{min} = 1$ second. We are currently modifying several common applications to take advantage of notifications.

The system shows negligible overhead. In particular, maintaining page ages has no more than 0.1% overhead for a scanning rate of once per second over the youngest pages.

We now examine the elasticity of applications to explore the potential of our system. A comprehensive evaluation under a variety of real workloads is underway, and will be included in a forthcoming full paper.

Mozilla cache: The Mozilla web browser maintains a cache of incoming web objects and pre-rendered pixmaps. We measure the loading time for a series of image-intensive web pages, starting from a cold cache and warm caches of small (10MB) and large (200MB) sizes. Using a small cache reduces the loading time by a factor of 3.5 due to the cached pixmaps, whereas a

large cache reduces it by an additional factor of up to 2. Hence when memory is plentiful, it is generally worthwhile having a large cache.

If the system encounters memory pressure, then the browser is able to release up to 110MB of memory. It is cheaper to page these objects to and from disk than to fetch them over the network; but due to the low hit rates typically observed for web pages, it is cheaper to simply free a large fraction of the cache. Therefore the application must factor in the cache hit rate when estimating the effective regeneration cost of its objects.

Malloc-held unused memory: The FreeBSD malloc implementation retains memory that is freed by applications to minimize the number of system calls, and avoid a performance degradation that we observed to be up to a factor of 8 in the worst case. However, with almost negligible penalty, it could release this memory to the system only when there is contention, and achieve significant gains. On average, on our workstations, application footprints contain at least 5% of malloc-held unused memory, and frequently 10% or more. Quantity-wise, the X server is the application with the most memory of this kind (up to 10 MB), while percentage-wise it is the window manager (almost always over 50% of its in-core memory).

5 Related work

There is a large body of work in virtual and physical memory management [3, 6, 4, 7]. To the best of our knowledge, our work is unique in that it enables applications to dynamically adjust their memory usage based on a metric provided by the kernel that quantifies the degree of memory contention, and thus the cost of using memory.

Page daemons in current OSs generally maintain only a rough LRU ordering of pages. They do not estimate the time of last access of individual pages, nor do they allow weighting of process memory allocations. In the working set model [3], the system maintains an estimation of each page's time of last access. This is done for a different purpose, namely, to estimate process working sets in order to avoid trashing.

In application-controlled file caching [2], applications are involved in controlling replacement in the OS file cache. Our work could be viewed as a generalization of this approach to the entire virtual memory.

Extensible kernel technology [1, 4] allows applications to modify operating system policies in a flexible manner. As such, sophisticated applications could use their extension interfaces to participate in physical memory management. Our system instead is aimed at enabling application-assisted physical memory management in general-purpose operating systems, via limited kernel changes and few new APIs.

The Nemesis OS provides memory isolation via *self-paging* [5]. Our system can also provide isolation, but is primar-

ily aimed at allowing elastic applications to exploit the available physical memory.

Recently, the VMware ESX server suggests a technique called *ballooning* to cooperatively manage memory between the OS and virtual machine guest OSs [7]. Our system notifies elastic applications about memory related events, but its goals and techniques are quite different.

The Mach OS supports *external paging*, where applications implement custom pagers invoked by the OS [6]. Our system also performs memory-related notifications, but it is different in that it conveys a severity metric to enables applications to adapt their memory use according to the cost of using this memory.

6 Conclusion

This paper proposes a memory management system that notifies applications about the level of contention for physical memory. Applications that have elastic memory requirements can react by adjusting their memory consumption in accordance to the current cost of using memory. This mechanism allows many applications to improve their performance by more aggressively allocating memory when there is no contention, while at the same time it improves system robustness by avoiding or reducing paging.

References

- [1] B. N. Bershad et al. SPIN—an extensible microkernel for application-specific operating system services. Technical Report 94-03-03, Dept. of Computer Science and Engineering, University of Washington, Feb. 1994.
- [2] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *USENIX Summer*, pages 171–182, 1994.
- [3] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.
- [4] D. R. Engler, M. F. Kaashoek, and J. O’Toole. Exokernel: An operating system architecture for application-level resource management. In *15th ACM Symposium on Operating System Principles*, Copper Mountain, CO, Dec. 1995.
- [5] S. M. Hand. Self-Paging in the Nemesis Operating System. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.
- [6] I. Subramanian. Managing discardable pages with an external pager. In *Proc. of the USENIX Mach Symposium*, Nov. 1991.
- [7] C. Waldspurger. Memory resource management in vmware esx server. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.