

Application-assisted physical memory management for general-purpose operating systems

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 later 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. This vastly simplified programming effort.

However, in practice, applications face a dramatic performance penalty if they perchance overflow the system's physical memory and induce paging activity. Applications are run in environments often unforeseen at development time, and are subjected to unpredictable work-

loads. They are therefore compelled to be conservative about their memory use, although they could often improve their performance by using more memory when it is available, and releasing it when there is memory pressure. Moreover, despite applications being conservative about memory usage, current systems (especially servers and older desktops) do face the risk of incurring a load burst that may result in severe paging. Some of this paging may be for application memory that could easily have been freed by the application with lower performance penalty than paging it out to disk and later paging it in. Therefore we contend that the virtual memory abstraction renders applications oblivious of the contention for physical memory, including those applications that can use memory pressure information to improve their performance and the system's robustness.

Our work provides feedback to applications about physical memory availability and the current cost of reclaiming memory. This is orthogonal and complementary to application-controlled caching [8, 12] and paging [22, 5, 10], which allow applications to control the page replacement policy and the mechanism for storing the contents of replaced pages. We treat paging as just one mechanism to reclaim physical memory; in fact, our system would be relevant and useful even if paging was hypothetically disabled. During normal operation without memory pressure, we encourage applications to step up their memory consumption, and try to exclusively use their adaptation methods to manage memory, as follows.

Many modern applications use algorithms that are *elastic* in their ability to trade memory consumption for performance or the usage of some other resource, like CPU, disk or network bandwidth. The following examples describe common applications that can achieve nontrivial performance gains using a substantially large memory footprint, but would prefer to release most of this memory than to page it out.

Language runtimes such as Java virtual machines perform garbage collection with substantially less overhead

if they use a large heap; in times of memory pressure they can quickly release this unused memory. Many malloc library implementations tend to retain the memory that applications free, so as to minimize their system calls; this memory can potentially be released to the system when needed. Databases, web caches and web browsers that maintain memory caches of disk and network data can increase their performance (often substantially) by enlarging their caches to very large sizes, or alleviate memory pressure by releasing a fraction of the cache that is cheaper to regenerate than page out. Moreover, databases, Java runtimes etc. can configure their footprints without need for manual configuration. Many scientific applications can trade CPU time for memory by retaining different amounts of partial results that may be reused at later stages in the computation. They often also have the ability to vary their problem size or simulation parameters (such as accuracy) in reaction to memory pressure, which is especially useful since many simulators are memory bound.

While it only takes small changes in applications to support these memory-related adaptations, it is nontrivial to decide when to invoke these adaptations, especially ones that have significant impact on the application's performance. This paper enables memory adaptations using the approach of informing applications about the severity of memory pressure using a carefully chosen metric. This would facilitate elastic applications to independently decide when and to what extent to perform their adaptations, thus consuming more or less memory as the situation demands. By taking advantage of elasticity in common applications, this system has the high-level goals of improving application performance, system robustness under severe memory pressure, and user control over memory allocations.

The rest of this section postulates our key ideas and motivates the problem and the solution approach. Then Section 2 proposes the severity metric. Using this metric, Section 4 describes a kernel-level page replacement framework using the metric. Section 5 presents results of an experimental evaluation, to demonstrate significant performance improvements achieved by common applications when memory is plentiful, and greater performance robustness in low-memory situations. Finally, Section 7 alludes to related work and Section 8 concludes.

1.1 Key ideas

Our scheme of system support for memory adaptation is based on the following four ideas:

(1a) Notifications: The system should *notify* applica-

tions 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) Severity metric: 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) Weighted adaptations: 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 and intuitive scheme of *preferences*, ultimately under user control, to determine which applications should first take actions and to what degree. For example, it should be possible to make background applications more aggressive about shrinking their caches than interactive applications.

(2b) Weighted page replacement: 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 be made more 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.

1.2 Motivation

Physical memory has become cheap and plentiful over the decades. There was intense debate in the 50's and

60's over memory management policies that avoid thrashing and minimize paging for scientific and business computing type workloads. Today, servers and even desktop machines are equipped with several GB of RAM, leading to the prevalent notion that memory shortages are easily eliminated by adding more memory, that systems only exhaust their memory if they are misconfigured, and that operating systems should not need to do much about memory management. However, we present key insights that motivated us to exploit elastic applications.

(1) *One can always envision constructive uses for more memory.* Currently, programmers are forced to remain conservative about memory usage. They can often improve performance by using more memory; for example, the OS can aggressively prefetch and cache disk blocks to substantially improve file system performance [21], especially if the system has cheap background I/O mechanisms like freeblock scheduling [15]. Section 5 presents several common applications that can achieve significant performance gains through high memory usage in the absence of memory pressure.

We envision future systems to operate in a regime where main memory is deliberately utilized to a high degree. While 1GB of RAM may seem excessive for current workloads, we expect that this memory will be almost fully consumed by various applications and the OS, on both servers and desktops. This is feasible only if applications are able to depend on a system facility that monitors memory pressure and informs them about when to allocate or release memory.

(2) *For memory adaptations to be the primary mode of memory management during normal system operation, they need to be graceful and continuous, and should frequently happen in numerous applications.* The key to gradual adaptations is a continuous severity metric that quantifies the cost of reclaiming memory, in a manner that applications can independently and gracefully react to, such as by varying its cache size by small amounts. Contrast this with a naive scheme of memory adaptation that avoids thrashing by notifying all applications each time there is memory pressure, and they aggressively react by freeing all the memory they can. Clearly, our graceful adaptation approach is preferable to this naive scheme if systems are to run at high memory usage, and are to support diverse adaptation methods with unequal performance impact.

(3) *Although computer systems usually have sufficient memory to avoid paging, it is always possible to encounter an unexpected load burst that can cause a disastrous performance degradation.* Workstations, laptops and various slim devices may have enough RAM (say 1GB) at the time of purchase, but from our experience,

this RAM often becomes insufficient after a couple of years of software installations and upgrades (especially of specialized applications with huge memory requirements). When systems become sluggish, users are accustomed to making them more responsive by closing windows or suspending applications; we contend that elastic applications should be exploited to make such intervention largely unnecessary.

(4) *It is well-documented that server administrators find it painful to manually adjust the footprints of server applications for high memory utilization without paging.* For instance, periodically tuning the size of database caches is performance critical, but tricky and time-consuming [23, 4]. There are extensive guidelines for assigning large heap sizes to Java virtual machines to reduce the frequency and overhead of garbage collection without paging [3]. If such applications are run standalone on the system, then one can assign a cache/heap size smaller than free memory available to applications – although conservatively, since free memory may decrease due to allocations by the kernel or system services. However, it is much worse if the database or JVM is co-located with other applications (which may be more instances of the same); then with dynamic workload fluctuations, it is painful to tune the memory settings and achieve the right balance. Our system allows diverse applications to independently configure their own footprints, and to dynamically adjust them in reaction to memory pressure, by normalizing their memory adaptation cost against a common reference point of potential paging cost.

Large-scale simulations are often memory intensive, and can influence their footprints by controlling their problem size. Users often go through painstaking iterations to guess a large enough problem size that will make the simulation just fit into memory. Instead, the simulation can be made portable across systems with different amounts of RAM, if it is designed to grow its problem size until notified of impending memory pressure, and then freeze the problem size and continue the simulation.

While these elastic applications can react to free memory availability information obtained by polling `top` or `vmstat` (this practice is followed in some specialized applications [3]), this information is insufficient for high memory utilization. The existence of the system's file cache often leaves little memory actually free, so the relevant information (provided by our severity metric) is the cost of reclaiming memory from the file cache, or by paging out application memory, or by freeing application buffers. Therefore, memory that was accessed a long time ago may never be accessed again, and may be released or even paged out with low performance impact to free up substantial memory for use of the application.

(5) *It is useful to extend CPU nice values to memory management.* There has been a recent surge of interesting background applications for desktop platforms such as peer-to-peer file sharing and web caching, and distributed computations like SETI@home and Folding@home. Shared computing platforms like Condor [6], and more recently, HPL’s Planetary Computing [19] and the Intel Research seeded PlanetLab [18] support guest applications with unpredictable memory requirements. Beyond isolating processes through explicit allocation limits and self-paging [11, 24], there has been limited work on control over memory allocation. Using our memory nice scheme, desktop users may specify that the working sets of interactive applications be retained in memory, while permitting background applications to use copious amounts of memory when foreground activity is less memory intensive. Shared hosting systems can use this by reactively nicing applications to achieve various QoS objectives.

(6) *Sometimes, users need to run applications dangerously close to the available memory, and wish for more assistance from the system in controlling the extent of paging.* Such scenarios are common in large-scale simulations, where users are familiar with having to monitor memory availability and paging I/O, and manually suspend and resume processes lest they thrash. Memory pressure notifications can be used to automate this, and voluntarily sleep at suitable moments.

2 Severity metric based on page age

What is the simplest OS primitive that can effectively and practically leverage the elasticity in applications, and achieve the diverse goals described above? We claim that the primitive is an appropriate choice of a single, well-defined, numeric memory pressure severity metric that the OS continuously reports to applications, coupled with a notification to all applications whenever the metric changes significantly.

We postulate four requirements of the metric.

(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 (expressed by extending CPU “nice” values to memory), by simply scaling the value of the metric.

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

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

2.1 Definition of the severity metric

We first define the *page age* of a physical memory page to be the time (in seconds) since it was last referenced. We use this to describe the computation of the severity metric in three scenarios of different levels of memory pressure.

A. When the system has exhausted its free memory: we define the system-wide *severity metric* for memory pressure to be the age of the least recently accessed (oldest) non-free page in the system. Note that a smaller value of the metric depicts a higher level of memory contention¹. The system may bound the metric by reasonable limits, say 5 seconds (S_{min}) and one hour (S_{max}).

When the system is paging, applications may wish to compare the paging cost against the cost of regenerating some memory that they can potentially release, to decide which alternative is cheaper. *This decision is inherently application-specific.* However, the following method of comparing amortized costs can serve as a guideline for many applications.

Assume for now that the system uses an approximately LRU-based page replacement policy. Then the severity metric is the age of the candidate page for eviction. Applications can use the LRU principle to assume that this page 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 expected paging I/O service time by the expected time to next access of the page (i.e., the severity). They can compare this paging cost against the total *regeneration 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, which could be the age of this chunk of 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

¹Despite smaller severity values indicating higher pressure, we still adopt this convention of reporting the age (rather than say its reciprocal), so that applications can deal with severity in convenient time units, and that our implementation is able to exponentially separate out larger severity values.

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.

B. When the amount of free memory is small: when free memory falls between zero and a system-defined threshold (say 10% of system memory), then the severity metric is gradually decreased from S_{max} to its target value of the age of the current page replacement candidate if paging were to occur. In particular, it is computed by geometrically interpolating between S_{max} (signifying no memory pressure) and the target value. 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 region of decaying severity metric and the corresponding early notifications is vital, since it tries to operate the system near full memory utilization without overflowing it. The gradual (rather than abrupt) decay of the severity metric serves three important purposes: (a) it allows applications some time to take actions, especially time-consuming ones like garbage collection. Applications that can quickly release memory can hold out until the severity metric decreases significantly below S_{max} . (b) it 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. Thus it provides the desired safety cushion. (c) it helps to avoid taking all such memory-releasing actions across the system all at once. Instead, a few of the cheaper and/or time-consuming actions may run first, and may free enough memory to relieve memory pressure and avoid taking more drastic actions.

C. When free memory is plentiful: paging is not likely in the near future, so the severity metric is set to S_{∞} , defined as $2 * S_{max}$.

Note that this metric is continuous to encourage adaptations to be gradual. Section 6 provides more rationale for this definition of the severity metric, by showing how it satisfies the four requirements better than several possible alternatives (such as available free memory or the volume of paging I/O).

3 Use of the severity metric

This section describes how our system uses the metric to leverage the elasticity in applications, perform nicing of

application actions, and preferentially page memory from different processes.

3.1 Application interface

Conceptually, the system always keeps all applications informed about the value of the severity metric, so that they may correspondingly adapt to try to maintain the invariant that the entire system (including all applications and the pager) operates at the *same level of severity*.

In practice, the system tracks the age of the page that is the current replacement candidate, and uses this as an approximation to the age of the globally oldest non-free page. It makes this metric continuously available to applications through a new system call named `severity`. A new signal named SIGMEM is dispatched to all applications each time the metric significantly changes, such as when it doubles or halves since the last notification. It is also dispatched every $\tau = 1$ second whenever the severity metric is smaller than S_{∞} , thereby enabling applications to take periodic actions.

Applications are free to ignore this signal. However, interested applications may implement signal handlers that query the severity metric upon signal receipt, and then potentially decide whether to release or allocate some memory based on logic described shortly. These signal handlers may reside in different layers of software, such as in application code (possibly even in the source code of high level programs), libraries, or language runtimes; in the latter two cases, the actual application may not need to be modified.

There is no need for a special API to express user-specified preferences at a process granularity: we simply extend Unix-style per-process CPU `nice` values to specify per-process weights for the memory subsystem. In the rare case that users wish to have different CPU and memory `nice` values, our system provides an overriding system call named `memnice` that sets only the latter.

We provide a resource utilization interface through a `resutil()` system call that reports the percentage utilization of CPU, disk and network interfaces over the past interval of say 10 seconds. Also, an `iocost()` function reports the average time taken for recent paging I/O operations to complete. This includes the disk queueing time for paging requests only if the queue length is more than a threshold (10 in our implementation); this is to remain conservative about our prediction accuracy over the rapidly fluctuating disk queue. These system calls may be used by some applications to derive a better estimate of paging I/O time, and to decide if an adaptation that affects some other resource is justified if that resource is under contention.

3.2 Application modifications

As mentioned earlier, the nature and extent of applications' reactions to memory pressure is based on their own discretion and policies. While they may completely ignore the notifications or even maliciously allocate more memory when there is pressure, it would generally be in their own interest to free some memory to avoid paging, and allocate more memory when it is available.

Hence, this section provides detailed guidelines that cooperative applications of different types may choose to follow. The following are four types of increasingly drastic actions, where the application may react in the described manner to the severity metric notification, in an attempt to achieve the global 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.

The malloc library can cheaply release the free pages it retains, so this action can be programmed to take place whenever the severity is anything below S_∞ , or in practice, whenever there is a notification for any value of severity.

Full garbage collection such as in JVMs is more expensive; moreover, it is usually not possible to determine the cost and yield of garbage collection beforehand (except approximately through statistics). Therefore, we trigger garbage collection only when memory pressure is slightly higher, when the metric is below a constant (10 minutes in our implementation). This will allow cheaper actions to be taken first, and will also ensure that old and presumably inconsequential pages lying in memory get released or paged out before garbage collection happens. Thus, the JVM would get a larger footprint at a relatively low cost.

(2) If memory releasing actions that are cheaper than paging 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 in-memory caches of file data, whose regeneration cost is trivially equal to I/O cost. They can release cache buffers that were accessed less recently than the severity metric, so that the buffers are prevented from unnecessarily being paged out to disk. Also, the system-wide LRU-approximation replacement policy is being naturally extended to application caches, contingent to alternative replacement policies that the application may employ.

In practice, the application may try to match its internal memory pressure with the system's memory pressure, say every time the severity metric doubles or halves, i.e., when SIGMEM notifications are dispatched. In the signal handler, rather than try to calculate the desired size of the cache, the application may enlarge or shrink the cache until the appropriate severity criterion is met. This makes the programming model for the database cache (or the modifications to existing databases) very simple.

(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. Then our guideline for cache replacement is to compare the amortized time-cost of paging against the amortized time-cost of freeing and regenerating these buffers, and to perform whichever is cheaper. This decision heuristic can be superimposed over any application-specific replacement policy, by expressing one in terms of the other, or by taking their conjunction.

(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. This may sacrifice some functionality or consistency, or may increase the consumption of some other resource, but should restore stability in the otherwise thrashing system. For example, a web browser may temporarily suspend image animations, or a database cache may choose to return stale or approximate results depending on what is acceptable to the application, or an image viewer like xv can show reduced quality images that the user can sharpen.

3.3 Extending "nice" to memory

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 page replacement. 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 nice-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 implementation of memory nicing for page replacement may enable 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 to assign *weights* to applications (where the weights are derived from the nice value of the process as explained in Section 4). Then drastic actions and recently accessed pages (e.g., code, static data) from niced-down processes can compete with mild actions and older pages (e.g., cache objects) from niced-up processes.

Application actions can be niced using the following simple and application-transparent mechanism. Rather than reporting the 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 niced-down process is informed that the severity is relatively higher, and is induced into taking more drastic actions.

Likewise, page replacement can also be subject to nicing, as follows. The amortized paging costs for a process may be amplified by linearly scaling down the age of all its pages by the weight of the process. *Page replacement is then performed in order of the resulting scaled ages*, and the globally reported severity metric is redefined as the maximum of the scaled ages of non-free pages. Such weighted page replacement would universally impose on all processes, regardless of whether they have been modified to use our API; so this age-based page replacement serves as an effective way to control rogue processes.

4 Design of the in-kernel system

The kernel portion of our design comprises of page age maintenance, severity calculation and notifications, and age-aware page replacement.

Page ages are computed while the OS scans page tables to inspect page reference bits. If the bit on an unreferenced page is found to be set, then the OS places this page in a *page bucket* containing pages of age between zero and $S_{min} = 5$ seconds. Every 5 seconds, the OS scans these pages, moves unreferenced pages to the second bucket with pages of age 5–10 seconds, and moves referenced pages back to the first bucket. Pages in the second bucket are slightly less likely to be accessed in the next 5 seconds, so the OS scans pages in this bucket every 10 seconds. In this manner, the OS maintains buckets with exponentially increasing age ranges until S_{max} , and scans pages in each bucket with the frequency of the age of the pages in that bucket.

These exponentially organized buckets cover the large age range of 5 seconds through 1 hour using only 10

buckets. The compromise is on the accuracy of ages: this scheme maintains ages correct to a factor of 2, and scans it just as frequently. The exponent of the bucket index multiplied by S_{min} gives the page age, so the OS does not need to measure timestamps while scanning or store per-page timestamps. For more fine-grained accounting of page ages, this ratio of 2 between buckets can be refined to say $\sqrt{2}$, which can be useful on memory intensive servers. A set of such age-based buckets is maintained for each process, and pages of that process are stored in these buckets.

Now that page ages are being efficiently maintained, the next goal is to select the page with maximum age, if the system wishes to perform LRU-based page replacement. For this purpose, buckets of the same index from different processes are *chained* together on one chain per bucket index. Selecting a page with the maximum age proceeds as follows. The system traverses these bucket chains from largest index downwards (right to left in Figure 1), and for each chain, traverses buckets in the chain to encounter lists of pages in that age range. A basic approximation to LRU replacement can evict pages in this order; other replacement policies can be implemented using guidelines provided in the next section.

Nicing processes: The next goal is to nice processes by scaling their page ages by some function of the nice value. For convenience, we internally associate Unix-style CPU nice values (typically in the $[-20, +19]$ range) with the memory subsystem. There needs to be a factor that determines how significant the impact of nicing is; this is S , designated to be the scaling constant (30 in our implementation). We exponentially scale the nice value onto the $[2^{-S/2}, 2^{S/2}]$ range of per-process weights, so that nicing (down) by +19 corresponds to scaling up the ages of the process by $W = 2^{S/2}$. On a general-purpose system, S would typically be a small multiple of the number of buckets per process B .

Conceptually, when a process is niced, the system needs to scale the process' page ages by this weight, and then reorder all pages in the system by their scaled ages for page replacement purposes. This can be made efficient by avoiding division and sorting operations, as follows. Adjacent buckets in any bucket list differ in their age range by a factor of 2. To scale up all of a process' pages by W , we simply have to shift each of its buckets $\log W$ chains to the right (refer to Figure 1). Observe that the exponent of the absolute index of the bucket after this shift times S_{min} is equal to the scaled age of the pages in the bucket, which is the *severity* of memory pressure if the page is the replacement candidate. Because nicing only involves detaching a small number of buckets per process ($B = 3$ in the figure and 10 in our implementa-

tion) from their respective chains and reattaching them in different chains, it is very efficient. This efficiency is central to our design, and is useful for purposes described in the rest of this section and in Section 4.1.

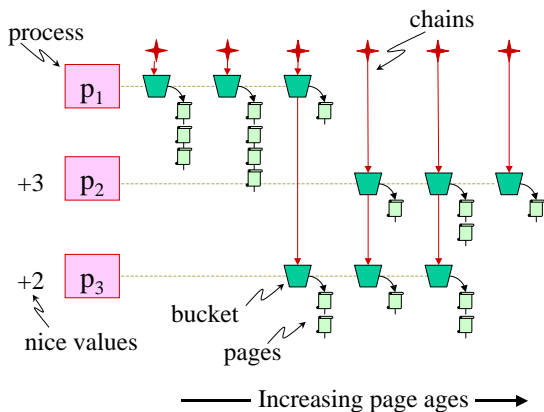


Figure 1: Maintaining pages in buckets.

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 a sufficient degree of nicing. These processes have therefore been consigned 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), and nicing by at least $\pm 40 * B/S = 14$ shifts the process to the next priority class.

In contrast, note that buckets of processes p_1 and p_3 in the figure overlap to some degree, so the recently accessed pages of p_3 compete with older pages of p_1 , achieving the type of overlap usually desired in practice.

Shared and unmapped pages: Page ages are scaled by the nice value of the process associated with the page. However, some pages are shared among multiple processes, whereas others belong to none. The latter consist of unmapped pages, and file cache pages that were accessed by the kernel on behalf of processes that access the file through the read/write interface.

We address the shared page issue by maintaining page ages at the granularity of *virtual page mappings* rather than physical pages. Then we define the scaled page age to be the smallest of the virtual page ages, scaled by the nice values of the respective processes. Age maintenance at the virtual page level is necessary because when a process is niced down (say because it is malicious), its shared pages reflect this new nice value.

We solve the unmapped page issue by associating such owner-less processes with special sets of *unmapped buckets*. Unmapping a page is an indication that the page may not be accessed again, so we *nice down* such pages by assigning them the nice value of the process that unmapped it, biased by a constant (1 in our implementation). This is analogous to (and more flexible than) FreeBSD’s approach of deactivating pages that it considers unlikely to be accessed again.

Finally, it is possible to implement a stripped-down and approximate version of memory pressure severity notifications, with nothing more of the design proposed above. While this is trivial to deploy, it has the disadvantage of not extending nice to memory management, nor being able to do so efficiently.

4.1 Uses of efficient nicing

Age nicing provides a versatile framework for three techniques briefly described here.

(a) Nicing-down clean pages

Clean pages are less expensive to evict than dirty pages, since they can be released without first writing them to disk; their I/O cost is only that of paging them in if accessed later. To prefer clean pages for eviction, we scale up the ages of clean pages by a factor of two; it is also conceivable to dynamically adjust the scaling factor to depend on the amount of transient disk contention. We believe this is a more systematic approach to distinguish clean and dirty pages than the second-chance schemes that OSs currently employ. In practice, we achieve this scaling by associating clean pages with a separate set of buckets and shifting them one position to the right.

Application buffers are generally dirty, so the ages of their buffers can be compared in the usual manner against the severity metric; however, if a buffer is known to be clean (e.g., cleanly memory mapped files or anonymous memory with a backing store copy), then the application may choose to scale up the age of such buffers by a factor of two to compare exactly against clean pages in the kernel.

(b) Specialized page replacement policies

This paper has, so far, described age-aware page replacement using an LRU-based policy. However, our system is fundamentally not tied down to any specific replacement policy. For example, it is possible to easily bias page ages by some function of the *frequency* of page accesses. This achieves a partial least-frequently-accessed

first policy, similar to activation counts in FreeBSD and page aging in Linux [17], but with finer control on the amount of bias. Again, efficient nicing is crucial for implementing this.

(c) Self-paging bias

Our design also facilitates system administrators to impart a *bias towards self-paging* [11], to ensure that processes needing memory mainly induce eviction of pages from their own footprint. This can be achieved as follows: whenever there is memory pressure, the system can determine which process needs memory at the moment. It could then temporarily nice-down that process by some bias specified by the administrator, perform page replacement, and restore its nice value.

We have not implemented this in our system, as we deem this feature to be inappropriate in general-purpose OSs. However, on special purpose OSs, such as those used for shared hosting environments, self-paging bias subsumes (and is more flexible than) simple memory isolation. A large self-paging bias (exceeding $\pm 40 * B/S$, from 4) would temporarily drop the memory requesting process into a lower priority class, thus achieving memory isolation of processes from each other (expected to be used in conjunction with suitable arbitration for allocating memory) [24]. Finally, a smaller value of the bias produces a novel effect of *partial isolation*, where the core working sets of processes are isolated, whereas older pages of all processes are made globally available.

To summarize, our proposed age-based memory management substrate facilitates system programmers to implement a wide range of policies, in a more flexible manner than current OSs typically allow.

4.2 Design challenges

We present three technical issues that this system faces in practice, along with corresponding solutions. These considerations are crucial towards making memory adaptations practical.

4.2.1 Priority inversion

When a process is niced down, its pages will be preferred for replacement even when they are more recently accessed than other pages in the system. This may increase the total paging I/O, and may induce some extent of priority inversion by degrading the performance of niced-up processes (as a result of being niced-up).

Solution: Nice for memory works best in combination with a proportional disk scheduler such as Cello [20] or YFQ [7] so that nice values can be extended into the disk subsystem.

4.2.2 Internal fragmentation

Application buffers may be smaller than physical pages. If some buffers on a page were recently accessed while most others are old, then the application may not wish to release such pages when notified of memory pressure. Coupled with the fact that it may have initially allocated a large amount of memory, this application may seem like a memory hog.

Solution: The application may enforce that its buffers be a multiple of page size (such as in the MySQL query cache); otherwise it may calculate buffer ages as the *average age* of the buffers in the container page, to allow for fair comparisons against the severity metric. It may also be more aggressive about shrinking if it detects internal fragmentation.

4.2.3 Double paging

The very act of an application releasing memory buffers may cause it to inadvertently page in some of that memory. For instance, a garbage collector may need to examine memory pages (possibly after paging them in) before deciding whether they are garbage.

Solution: We request applications that may face this issue to consider using the *mincore* system call to determine if a set of pages is in main memory or backing store, and to try to account for the potential page-in cost before accessing or releasing the memory.

5 Evaluation

We implemented the *severity* system call and the SIGMEM notifications in the Linux-2.4.4 and FreeBSD-4.3 operating systems. In the latter case, we also implemented our age-aware page replacement strategy so that nice could be extended to memory management.

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 $S_{min} = 5$ seconds.

We now examine several elastic applications to demonstrate the performance gains achieved by being less con-

Memory consumption	Runtime
220 MB	240.7 sec
225 MB	249.3 sec
227 MB	255.7 sec
228 MB	506.8 sec
230 MB	817.7 sec
231 MB	never completes

Figure 4: Runtime spike in the 1MB case due to thrashing.

constant, set to 10 minutes in our implementation (as described in Section 3.2). Thus, in times of memory pressure, some old pages in the system get evicted first, and then garbage collection occurs. If the severity remains high, then the notification arrives periodically every second, thus triggering garbage collection sufficiently often.

We set the JVM heap size to a very large value (10GB), and allowed the severity notifications to control it exclusively through garbage collection. Under normal execution, the heap grew till 222MB and 219MB in the 100KB and 1MB experiments respectively, and then periodic garbage collection maintained the system close to full memory utilization. The entire experiment runs in less than 10 minutes, so when the severity metric for the JVM pages is compared against garbage collection, the latter gets preferred over paging out parts of the JVM. *This eliminates the need for manual heap size configuration, making the simulation portable across machines with different amounts of RAM.*

But operating close to available RAM might be dangerous if another application were to suddenly consumed a lot of memory. To study this, we ran a memory hog that allocated and actively accessed 90MB of memory. Without our severity adaptations, the 100KB simulation degrades in performance by 150% (runtimes depicted in Figure 5), and the 1MB simulation thrashes as expected. With the severity notifications and the garbage collection handler, the JVM reacts to the memory hog by gracefully decreasing its footprint till both applications are accommodated in RAM. As a result of using a smaller heap, the two simulations suffer a slowdown of only 25% and 55%, and the system does not thrash. *This demonstrates how our system enables applications to be robust under memory pressure, while improving their performance when free memory is available.*

5.1.3 Nicing-down memory hogs:

As a final experiment, we niced-down the memory hog process by 14, which is just enough to push it into the next priority class (see Section 4). Then the JVM ran with

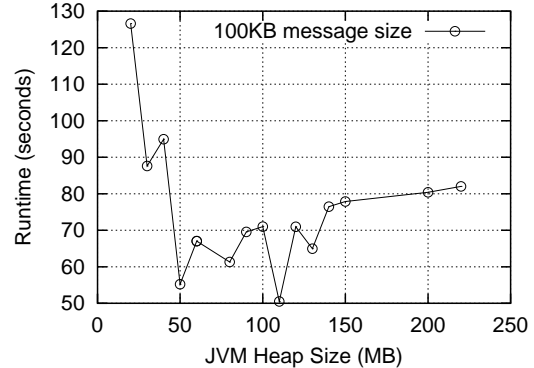


Figure 5: JVM running times with memory hog.

heap set to the full 220MB it normally consumes, and the memory hog replaces memory from its own footprint. Thus, the effects of this rogue process are isolated from the JVM. This hog application accesses memory rapidly, so nicing it down by a small amount (less than 14) has little or no effect.

Section 4.2.1 alludes to priority inversion possible when a process is niced, say because the memory hog process causes disk interference with the JVM. This problem is not observed because the JVM does not perform much disk I/O – neither file I/O nor paging. Even if it does perform I/O, our proportional disk scheduler would ensure that it receives a fair share of disk service, and would not be starved due to paging I/O from the memory hog. It would still suffer a slight latency degradation because disk I/O cannot be perfectly isolated, but this effect can be reduced using a combination of modern disk scheduling techniques.

5.2 Mozilla cache

This experiment demonstrates the applicability of memory adaptation to desktop environments. The Mozilla web browser maintains a memory 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. For real web client workloads [14], benefits acquired through caching may vary depending on browsing patterns, and on whether the working set is closer to 10MB or 100MB. In general, when memory is plentiful, it is worthwhile using a large cache.

We implemented the memory resizing adaptation in Mozilla by partially taking the actions that “clear mem-

ory cache” does. Mozilla has some internal support for reacting to memory pressure: when its memory module detects allocation failures, or when it triggers a low memory condition (which it never does on Unix systems), then other subsystems become more conservative about memory usage. Since there is no external interface connecting Mozilla to the system’s memory management unit (such as a severity metric notification), this aspect of Mozilla is underdeveloped, and so we did not use it.

If the system first encounters memory pressure (severity drops from S_∞ to S_{max}), then a severity notification is received, and the browser shrinks its cache to a reasonable value of 30MB. Upon increasing memory pressure, the browser is able to release an additional 20MB of memory with modest performance degradation. It is actually cheaper to page these objects to and from disk than to fetch them over the network (milliseconds versus seconds); however, due to the low hit rates typically observed for web pages, our adaptation finds it cheaper to simply free a large fraction of the cache, bringing it down to 10MB as long as there isn’t severe and sustained memory pressure. Therefore the application must factor in the cache hit rate while estimating the effective regeneration cost of its objects.

5.3 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 can 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 pages, 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). A simple SIGMEM handler that releases these unused pages can be embedded into libc, and dynamically linked applications need not be changed.

5.4 Overhead measurement

Our in-kernel design efficiently performs three key operations: (1) maintaining page ages, (2) selecting max-weighted-age page for replacement, and (3) nicing processes for memory. The only appreciable overhead encountered by our system is while scanning pages for noting page reference bits. We now characterize this over-

head.

Our system scans pages only as frequently as their age itself, so the common-case overhead of scanning would be quite low. Indeed, on a system with 256MB RAM running memory intensive applications such as a kernel build and a Mozilla web browser, the overhead of our system is as low as 0.1%.

To exercise the pathological case, we measure the time it takes to scan all the pages on the system once. This includes scanning free pages and kernel pages, and is therefore more than the maximum possible overhead. This scanning is observed to take 40ms, so if it is performed as often as once every $S_{min} = 5s$, then the overhead is 0.8%. This scales linearly with more RAM, so on a 1GB RAM machine, this pathological case overhead is 3.1%.

We perform page scanning even without memory pressure, but under normal operation, pages would rarely have ages of a few seconds, and with plenty of free memory, the free pages would not be scanned. Therefore we deem it okay to scan all the time, with the advantage that a sudden burst of memory pressure would encounter a state where page ages are known and sensible actions can be taken. Finally, if the overhead is found to be large with many GB of RAM, it is always possible to gracefully compromise accuracy for efficiency by increasing S_{min} .

6 Discussion

6.1 Trust and incentive

We assume that applications cooperate with the OS to improve overall system performance. Faulty, greedy, or malicious applications can at worst increase memory pressure by allocating too much memory and ignoring memory pressure notifications. This is essentially no different from current general-purpose OSs, and can be prevented by extraneous mechanisms such as performance isolation [24]. Moreover, our system enables users to isolate such rogue applications by nicing them down with respect to memory (by scaling up their page ages), which we describe in the next section.

It is fundamentally impossible for the system to distinguish a malicious application from a genuine one with high memory demands that the user prefers to meet. Therefore, other than this ability to nice processes for memory, we do not try to incorporate any isolation mechanisms or strategy-proof decisions into the OS. In the same spirit, we also aim for overall performance by presenting applications with a global severity metric, rather than per-process fairness by normalizing the metric for

each process to some baseline. For the latter, we simply rely on nicing.

The main incentive for a programmer to modify applications for our system even when other applications might be non-cooperative is that it allows the application to safely use copious amounts of memory without the risk of paging, or in the long term, being niced down for being non-cooperative.

6.2 Deployment strategy

This section examines the migration path between current systems and a world where elastic applications are distributed with appropriate hooks. Firstly, the deployment can be incremental. Programmers can independently code small hooks into elastic applications, which are innocuous if the system does not have severity support. Otherwise, the hooks control the allocation and release of resizable memory regions. Not all applications need to support these hooks; however, benefits accrue to both the cooperating application as well as the system when more and more applications (and libraries, modules, etc.) support them.

Secondly, these hooks are small and easy. From our experience in modifying applications, it involves identifying resizable memory regions, and the manner of resizing them. Sometimes fixed-size regions (with size specified at startup) have to be recoded to be dynamically resizable if possible. Then implement a signal handler that either enlarges or shrinks the cache until the severity condition is met. Some adaptations may need to explicitly profile its regeneration cost in time units; also, for caches, it may be useful to measure their average hit ratio and calculate the effective regeneration cost by multiplying the two. For a typical adaptation, we found that about 10–100 lines of code and an afternoon of instrumentation time suffices.

7 Related work

Memory management is an old problem, and is now nearly irrelevant due to the abundance of physical memory. Yet, there *is* a fundamental problem in that memory is difficult to manage and application-held memory is largely unmanaged by the OS. If and when this problem ever materializes in the form of memory exhaustion and thrashing, the consequences are fatal. There is a circular chain of reasoning that keeps current systems generally away from this disaster. Programmers are conservative about memory consumption (at the cost of performance), and users and administrators exert effort to man-

age memory. Automating memory management and extracting high performance through liberal memory consumption has always been an unattainable goal due to insufficient support from both sides of the system – the OS and the applications – and the absence of a simple primitive that facilitates this synergy across diverse application types. We have provided this link through our novel severity metric, and have realized the potential to solve this long-standing problem of general-purpose memory management using application adaptations.

This section describes past efforts along the three directions that converge into our solution, namely OS and application support for memory adaptations, and OS support for policy control on memory allocation.

OS support for memory adaptations

There is a huge body of work in application-controlled page replacement mechanisms and policies [22, 11, 5, 10, 8, 12]. Our work is distinct from the above in that it is orthogonal and complementary to paging. 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 enables comparison of the costs of reclaiming memory from different sources. Now we compare our work in more detail against some of these other systems.

In different schemes for *external page cache management* [8, 12], applications are involved in controlling replacement in the OS file cache. Our work can be viewed as a generalization of this concept to application memory. Our key insight is unique from the above two: [8] uses a two-level scheme for allocating memory in the kernel and deciding replacement policies in applications, and [12] takes the approach of exposing page faults to applications, whereas we convey a metric that quantifies memory pressure and take independent decisions based on that.

Extensible kernel technology [5, 10] 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* [11]. Our system can also provide isolation between processes in terms of memory; in fact, as Section 4 explains, we also enable partial isolation through an adjustable self-paging bias. However, our system is pri-

marily aimed at allowing elastic applications to exploit the available physical memory, and the isolation aspect is considered an essential secondary feature.

Recently, the VMware ESX server suggests a technique called *ballooning* to cooperatively manage memory between the OS and virtual machine guest OSs for shared hosting [25]. Our system also notifies elastic applications about memory related events, so in that sense there's a similarity; however, we target a mostly cooperative setting in a general-purpose OS (rather than treat applications as black or gray boxes like in VMware), and our goals and techniques are quite different.

The Mach OS supports *external paging*, where applications implement custom pagers invoked by the OS [22]. 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.

The Solaris 2.0 operating system provides many features for advanced memory management. For instance, it provides detailed guidelines for manually analyzing memory pressure, by interpreting the output of *vmstat*, *memtool*, etc. [16]§7.1.5. It does not automate this memory pressure analysis, nor does it explore application reactions to it. Furthermore, the criteria used do not quantify the age of pages in the system, and are not useful for adaptations.

Application support for memory adaptations

In the opposite direction, elastic applications have always needed some means of determining how large to set their configuration parameters.

Some applications rely on the programmer's and the user's judgement to calculate this size. For example, there is considerable literature on the web on tuning the size of database caches [23, 4]. While co-locating the database with other applications such as web servers, this size needs to be chosen (and re-chosen) carefully. A documented rule of thumb is to calculate the hit ratio of the cache for each tentative value of the cache size, and keep enlarging it until the hit ratio curve flattens out. With our system, the database can be made to automatically tune its query cache to a size where the oldest query result in the cache has slightly higher cost of freeing and regenerating than the cost of paging I/O. This approximates the optimal cache size.

Likewise, there are extensive guidelines for setting JVM heap sizes in applications such as the BEA WebLogic server [3]. The rule of thumb is to minimize the time spent performing garbage collection, while maxi-

mizing the number of clients it is able to support. Therefore, the heap size may be conservatively set to 80% of system memory. Furthermore, the heap size needs to be manually observed before and after full garbage collections; if the heap size does not change by much, then the heap size setting may be decreased to increase the number of clients or to benefit other applications, without excessively impacting the garbage collection overhead. Garbage collection may also be performed within the young heap (generational GC) for taking advantage of cache locality. Interactive applications may also collect when the user is experiencing periods of idle time. Our system does *not* insist that all these garbage collection heuristics be ignored: in fact, generational collection, idle time collection, etc. are always beneficial. However, whenever these cheap collections prove insufficient, our system enables full garbage collection to trigger at key moments, namely when there is memory pressure of nontrivial severity. Thus, manual intervention for tuning JVM heap sizes can be almost completely made unnecessary.

Some other applications attempt to provide internal support for adapting their footprint, and inevitably run up against the lack of system-wide knowledge. For example, as Section 5.2 describes, the Mozilla web browser has a partially developed scaffolding for releasing cache memory from various subsystems (through an internal notification scheme) when there is memory pressure. However, on Unix systems, the only means of triggering this is to watch for malloc failures – which happen upon exhaustion of virtual memory or upon reaching a resource limit. On Windows systems, it also watches for the free memory pool to reach 5%, but this ignores the existence of the file cache, and cannot capture degrees of memory severity. In our system, we can simply hook this internal memory adaptation scheme to a SIGMEM handler. However, this support in Mozilla is currently immature, so we have not used it.

Policies for memory allocation control

Traditionally, memory in general-purpose operating systems has been allocated to applications on a need basis. Proportional scheduling paradigms for other resources have been adapted to memory [24, 11] in the form of isolation or partitioning. However, they encounter the obstacle of not knowing the appropriate partition size, and having to decide it manually or through some situation-specific policy. Isolation also does not cater to applications' memory demands, since it fails to distinguish between memory allocations for needy and almost-idle processes. Therefore practical uses of memory isolation are limited.

The VMware ESX server uses memory isolation between virtual machines, but proposes a concept called *idle page tax* [25]. At a high level, it inversely scales the memory proportions of each virtual machine by a statistically estimated fraction of its pages that have not been accessed for a long time. Thus, the apparent difference of this estimate from our severity metric is that of the fraction of old pages versus the age of the oldest page.

Interestingly, each of these two metrics is appropriate to the corresponding problem. In the VMware case, the OS wishes to recover pages from the virtual machine that holds a lot of idle memory. The use of this page may depend on the guest OS's internal use of the page, so the OS uses a combination of ballooning to trigger the guest OS's page replacement policy, and forced pageout in times of severe pressure. On the other hand, our system has the cooperation of applications, and aims to release the *oldest page in the system*, to maintain a balanced level of severity across applications. Beyond that it hands over the choice of freeing the page to the application's policy.

In [2], web servers are adaptively allocated resources to meet a high-level throughput criterion. They are subject to an initial admission control phase, so when they are online, they are partitioned the same amount of memory as initially planned. Over time, their characteristics are studied, and these partitions adjusted through estimations of their working sets.

This non-cooperative scenario does not directly lend itself to our memory adaptation system. Yet, there is use for a partial isolation scheme on the lines of what Section 4 proposes. Then the core working sets of the web servers are retained in memory and isolated from each other, while old memory pages are gracefully shared between all web servers to tolerate bursty workloads and smoothen the periods between partition readjustments. The age-based severity metric is the key to providing partial isolation.

Solaris provides a feature known as *priority paging*, which is mainly applicable to shared code in dynamic libraries [16]§5.8.4. This is different from the memory nicing mechanism we propose in this paper.

Other virtual memory APIs

There is literature on OS support for garbage collection, mostly in the form of remapping or unmapping pages that have been swapped out. The Symbolics 3600 architecture closely intertwined GC with virtual memory, and the AIX `disclaim` system call can be used for similar purposes [13]§ 6.5. Our system provides a different kind of OS support for garbage collection, geared towards heap size management.

Appel and Li [1] identify a set of OS virtual memory primitives that can be exported to several classes of interested applications for considerable performance benefit. These include concurrent and generational garbage collection, shared virtual memory, concurrent checkpointing, persistent stores, extending addressability, data-compression paging, and heap overflow detection. These primitives are pertinent to memory protection, whereas our system proposes a primitive that exposes memory pressure.

8 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. It minimizes the need for manual footprint configuration, and enables a means of extending "nice" to memory management.

References

- [1] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *ASPLOS*, volume 26, pages 96–107, New York, NY, 1991. ACM Press.
- [2] M. Aron, S. Iyer, and P. Druschel. A resource management framework for predictable quality of service in web servers, July 2001. Submitted. <http://www.cs.rice.edu/~ssiyer/r/mbqos/>.
- [3] BEA WebLogic Java heap size tuning guide. <http://edocs.bea.com/wls/docs70/perform/JVMTuning.html>.
- [4] Berkeley DB Reference Guide: Selecting a cache size. http://www.sleepycat.com/docs/ref/am_conf/cachesize.html.
- [5] 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.
- [6] A. Bricker, M. Litzkow, and M. Livny. *Condor Technical Summary, Version 4.1b*, 1992.
- [7] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with Quality of Service guarantees. In *ICMCS, Vol. 2*, 1999.
- [8] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *USENIX Summer*, pages 171–182, 1994.

- [9] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), Oct. 2002.
- [10] 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.
- [11] S. M. Hand. Self-Paging in the Nemesis Operating System. In *3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.
- [12] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proc. of the 5th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, Boston, MA, Oct. 1992.
- [13] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic dynamic memory management*. John Wiley and Sons, Inc., New York, NY, 1996. ISBN 0-471-94148-4.
- [14] T. Kelly. Thin-client web access patterns: Measurements from a cache-busting proxy. *Computer Communications*, 25(4):357–366, Mar. 2002.
- [15] C. Lumb, J. Schindler, and G. Ganger. Freeblock scheduling outside of disk firmware. In *Proc. of the USENIX Conference on File and Storage Technologies*, Monterey, CA, Jan. 2002.
- [16] J. Mauro and R. McDougall. *Solaris Internals: Core Kernel Architecture*. Pearson Education, 2000. ISBN 0-13-022496-0.
- [17] Page replacement in Linux 2.4 memory management. <http://www.surriel.com/lectures/linux24-vm.html>.
- [18] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. of the 1st Workshop on Hot Topics in Networks*, Princeton, NJ, Oct. 2002.
- [19] J. R. Santos, K. Dasgupta, G. Janakiraman, and Y. Turner. Understanding service demand for adaptive allocation of distributed resources. In *Proc. of the IEEE Globecom Global Internet Symposium*, Taipei, Taiwan, Nov. 2002.
- [20] P. Shenoy and H. M. Vin. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1998.
- [21] E. Shriver, C. Small, and K. Smith. Why does file system prefetching work? In *Proc. of USENIX 1999 Annual Technical Conference*, pages 71–84, Monterey, CA, June 1999.
- [22] I. Subramanian. Managing discardable pages with an external pager. In *Proc. of the USENIX Mach Symposium*, Nov. 1991.
- [23] Tuning MySQL Server 4.0 Query cache. <http://www.mysql.com/newsletter/2003-01/a0000000108.html>.
- [24] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared memory multiprocessors. In *ASPLOS*, Oct. 1998.
- [25] 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.