Transparent Contribution of Memory

James Cipar Mark D. Corner Emery D. Berger*

Department of Computer Science
University of Massachusetts-Amherst, Amherst, MA 01003
{*jcipar, mcorner, emery*}@*cs.umass.edu*

Abstract

A multitude of research and commercial projects have proposed *contributory* systems that utilize wasted CPU cycles, idle memory and free disk space found on end-user machines. These applications include distributed computation such as signal processing and protein folding, peer-to-peer backup, and large-scale distributed storage. While users are generally willing to give up unused CPU cycles, the use of memory by contributory applications deters participation in such systems. Contributory applications pollute the machine's memory, forcing user pages to be evicted to disk. This paging can disrupt user activity for seconds or even minutes.

In this paper, we describe the design and implementation of an operating system mechanism to support transparent contribution of memory. A *transparent memory manager* (TMM) controls memory usage by contributory applications, ensuring that users will not notice an increase in the miss rate of their applications. TMM is able to protect user pages such that page miss overhead is limited to 1.7%, while donating hundreds of megabytes of memory.

1 Introduction

A host of recent advances in connectivity, software, and hardware has given rise to *contributory systems* for donating unused resources to collections of cooperating hosts. The most prominent examples of deployed systems of this type are Folding@home and SETI@home, that donate excess CPU cycles to science. Other examples include peer-to-peer file sharing applications that enable users to donate outgoing bandwidth and storage and receive bandwidth and storage in return. The research community has been even more ambitious, proposing systems that harness idle disk space to provide large-scale distributed storage [8, 4, 3]. All of these applications, be they primarily processing or storage, require a donation of idle memory, since contributory applications consume memory for mapped files, heap and stack, as well as the buffer cache.

Users are only willing to participate in such systems if contribution is transparent to the performance of their ordinary activities. Unfortunately, contributory applications are not at all transparent, leading to significant barriers to widespread participation. Contributing processing and storage leads to memory pollution, forcing the eviction of the user's pages to disk. The result is that users who leave their machines for a period of time can be forced to wait for seconds or even minutes while their applications and buffer cache are brought back into physical memory. In this paper, we show that this figure can grow to as high as 50% degradation after only a five minute break.

A number of traditional scheduling techniques and policies, such as proportional shares [10], can prevent only some kinds of interference from contributory services. For instance, if the owner is not actively using the machine, a contributory service can use all of the resources of the machine. When the user resumes work, the resources are reallocated to give fewer resources to the contributory service. However, while the resource allocation of a network link or processor can be changed in microseconds, faster than any user can notice, memory allocation does not work well with the same strategies. Due to the reliance on relatively slow disks, the memory manager can take minutes to page while the user waits for an unacceptable amount of time.

As a solution to this problem we present the *transparent memory manager* (TMM), which protects *opaque* applications from interference by *transparent* applications. *Opaque* applications are those for which the user is concerned with performance. Typically these will be all of the applications run for the user's own benefit on

[*] This material is based upon work supported by the National Science Foundation under CAREER Awards CNS-0447877 and CNS-0347339. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

their workstation. Such opaque applications may be interactive, batch, or even background applications, but users prioritize opaque use over contributory applications. *Transparent* applications are those which the user is running to contribute to a shared pool of resources. TMM ensures that contributed memory is transparent: opaque performance is identical whether or not the user contributes memory.

A key feature is that TMM is dynamic: it automatically adjusts the allocation given to opaque and transparent applications. This is in sharp contrast with schemes like Resource Containers that statically partition resources [5]. Such static allocations suffer from two problems. First, a static allocation will typically waste resources due to the lack of statistical multiplexing. If one class is not using the resource, the other class should be able to use it, and static allocations prevent this. Second, it is unclear how users should choose an appropriate static allocation—as the workload changes, the best allocation changes drastically. Third, the primary concern of users is the effect that running a service has on the performance of their computers. It is not possible for a user to determine how that translates into an allocation.

Another important property of TMM is that it is a global policy. It considers the behavior of all opaque applications together when determining the limits, and limits to total memory use consumed by all transparent applications. System calls such as madvise can be used by individual applications to limit their own impact of system performance, but many transparent applications running concurrently would have to coordinate their use of madvise to ensure that their aggregate memory use did not exceed the limit. Furthermore, the necessary modification of application code is contrary to our design goals.

The rest of the paper describes the design in Section 2, implementation in Section 3, and evaluation in Section 4. Our evaluation shows that TMM is able to limit the effect of transparent applications to a 1.7% increase in page access times while allocating hundreds of megabytes to transparent applications. An expanded version of this work, including a mechanism for transparent storage, is available in a technical report [2].

2 Design

The goal of Transparent Memory Management (TMM) is to balance memory allocations between classes of applications. TMM allocates as much memory as it can to contributory applications, as long as that allocation is transparent to opaque applications. The insight is that opaque processes are often not using all of their pages profitably and can afford to donate some of them to contributory applications. The key is to decide how many pages to donate and to donate them without interfering with opaque applications.

Here we provide an overview of TMM: (i) it samples page accesses with a lightweight method using page reference bits, (ii) it determines allocations using an approximate Least Recently Used (LRU) histogram of memory accesses based on the sampled references, (iii) it evicts opaque pages in an approximate LRU manner to free memory for transparent applications, (iv) it ages the histogram to account for changing workloads, and (v) it filters out noise in page access to keep the histogram constant even when the user takes a break. We omit the details of (i) and describe the rest in more detail below.

2.1 Determining Allocations

The key metric in memory allocation is the access time of virtual memory. As TMM donates memory to transparent tasks, opaque memory will be paged out, increasing the access time for opaque pages. Thus TMM determines the amount of memory to contribute by calculating what that allocation will do to opaque page access times. The mean access time (MAT) for a memory page is determined from the miss ratio (μ), the time to service a miss (m), and the time to service a hit (h):

$$\text{MAT} = \mu \cdot m + (1 - \mu) \cdot h. \tag{1}$$

If the system allows background processes to use opaque pages, it will increase the miss rate μ of opaque applications by a factor of β, yielding an increase in MAT by a factor of:

$$\frac{\text{MAT'}}{\text{MAT}} = \frac{\mu \cdot \beta \cdot m + (1 - \mu \cdot \beta) \cdot h}{\mu \cdot m + (1 - \mu) \cdot h}. \tag{2}$$

In the case of a page miss, the page must be fetched from the page's backing store (either in the file system or from the virtual memory swap area), which takes a few milliseconds. On the other hand, a page hit is a simple memory access, and takes on average just a few tens of nanoseconds. Because these factors differ by many orders of magnitude, TMM can estimate that the average page access time is directly proportional to the number of page misses. Increasing the miss rate by β will make the ratio in (2) approximately β. This ratio is valid as long as there are some misses in the system. TMM uses a value of $\beta = 1.05$ by default, assuming that most users will not notice a 5% degradation in page access times.

2.2 LRU Histogram

Limiting an allocation's effect to the factor β requires knowing the relationship between memory allocations and the miss rate. This relationship can be directly determined using a Least Recently Used (LRU) histogram, also known as a page-recency graph [9, 11, 12] or a stack distance histogram [1]. An LRU histogram allows TMM to estimate which pages will be in and out of core for any memory size. Using this histogram, TMM can determine what the miss rate for opaque processes would be for any allocation of pages to transparent processes. Building the histogram depends on sampling page references, and we use a method based on reference bits [2]. Previous research shows that these sampling and approximation-based approaches work well even as many operating systems use approximations of LRU, such as CLOCK [12], or 2Q [7].

In an LRU histogram H, the value at position x represents the number of accesses to position x of the queue. Thus $\sum_{i=0}^{x}[H(x)]$ is the number of accesses to all positions of the histogram up to and including x. This value is approximately equal to the number of page hits that would have occurred in a system that had a memory size of x pages. Subtracting this value from the total number of accesses in the workload gives the number of misses for that memory size. It is important to note that the LRU histogram contains all of the *virtual* pages in the system. With only the physical pages, it would not be able to predict what the miss rate would be given more memory pages. A sample cumulative histogram and memory allocation is shown in Figure 1.

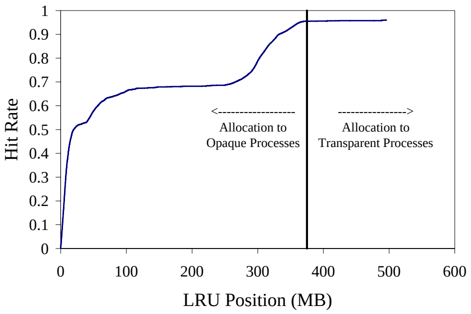

Figure 1: This figure demonstrates a sample histogram and allocation. In this case, the user has a working set size of approximately 390MB and can afford to donate the rest of the physical memory.

2.3 Page Eviction

When applications allocate pages, the operating system will first determine if there are free pages. If there are, it simply hands them to the process, regardless of the limits determined from the histograms—there is no reason to deny use of free pages. However, when free pages run low, the OS will force the eviction of other pages from the system and must choose between transparent and opaque pages. The choice depends on the limits determined by the factor β, and the current allocation of pages in the system. If both opaque and transparent applications are above their limits, or both are below their limits, it favors opaque applications and evicts transparent pages. Otherwise it will evict from whichever class is above its limit.

2.4 Aging the Histogram

We age the histogram over time using an exponentially weighted moving average. TMM keeps two histograms: a permanent histogram that TMM computes limits from, and a temporary histogram that records only the most recent activity. After some amount of time t, we first divide the temporary histogram by the total number of accesses which happened in that aging period so that the values represent hit to miss ratios. We then add the temporary data into the permanent histogram using an exponential weighted moving average function and then recompute the cumulative histogram. We adopt a common value $\alpha = \frac{1}{16}$ and adjust the time t to match this.

The difficult part lies in tuning t. If TMM is not agile enough (too stable), rapid increases in opaque working set sizes will not be captured by the histogram, transparent applications will be allocated too many pages, and opaque page access times will suffer. If TMM is not stable enough (too agile), infrequently-used opaque applications will not register in the histogram and may be paged out.

To deal with this, we adopt a policy that is robust but favors opaque pages. Normally, t is set to 10 minutes for stability. This value is large enough that applications used in the last day or two bear enough significance in the histogram to force the memory limit to not page them out. However, to remain agile, the system must move the limit in response to unusually high miss rates. If TMM notices that the page misses have violated the stated goal, it adopts a more agile approach, using the most recent sample as the temporary histogram and immediately averaging it with the stable one. This policy has the disadvantage of stealing pages from transparent applications based on transient opaque use, but it is required to favor opaque applications over transparent ones.

2.5 Dealing with Noise

The goal in aging the histogram is to detect phase shifts in user activity over time. When the user is not actively using the machine, the histogram should remain static, directing TMM to preserve the opaque pages in the cache. However, we have observed that even when not actively using the machine, our Linux installation still incurs many page references from opaque applications. If left long enough, TMM misinterprets these accesses as shifts in user behavior and will change the allocation of pages to opaque applications. As there are only a small number of these pages, when the user is not using the system, it appears that the working set has very high locality. TMM will act improperly, allowing transparent allocations to consume a large number of pages in the system.

To avoid this, TMM filters its page samples. To decide whether or not to age the histogram after an aging period of t seconds, we only consider the opaque page accesses that have touched the LRU queue at a point farther than 10% of the total. If there were more than ten such accesses per second, we age the histogram. This policy also implies that we need to guarantee a minimum of 10% of the system's memory to opaque application, a reasonable assumption. We found that the idle opaque activity rarely touches pages beyond the 10% threshold. We observed that with filtering, TMM never ages the histogram during periods of disuse.

3 Implementation

TMM requires a number of kernel modifications as well as several user-space tools [1]. Inside the kernel, we implemented tracing methods that periodically mark pages as unreferenced, and then later test them for MMU-marked references. The list of page references and recent evictions are passed through a `/dev` interface. A user space tool written in C++ reads these values and tracks the LRU queue. This tool computes the transparent and opaque limits and passes the limits back into the kernel through a `/sys` interface.

Additionally, we have augmented the Linux eviction policy kernel with our LRU-directed policy which enforces memory limits. Another user-space tool, `maketransparent`, allows users to mark processes as transparent. All pages that the transparent processes use are then limited using TMM. Pages that are shared between opaque and transparent pages are marked as opaque pages, but to ignore noise caused by transparent process access, hits to those pages are not traced.

4 Evaluation

In evaluating TMM, we sought to answer the following questions: (i) How well does TMM prevent transparent processes from paging opaque memory out, and how does TMM's dynamic technique compare to static allocation? (ii) What is the transient performance of TMM? (iii) What is the overhead in using TMM?

The primary function of TMM is to ensure that contributory processes that use memory do not interfere with the user's applications. To show TMM's benefits, we simulate typical user behavior: we use several applications, take a coffee break for five minutes, and return to using similar applications. During the coffee break, the machines runs a contributory application. As a contributory application, we use a program called POV-Ray, a widely-used distributed rendering application. The rendering benchmark we used with POV-Ray caused it to have a working set size greater than the physical memory on our test platform. Most contributory applications do not use memory this aggressively; POV-Ray could be considered a "worst-case" contributory application. For this experiment, we compare five systems with three different sets of opaque applications. The five systems are as follows: vanilla Linux, TMM with three different static allocations for opaque applications (25%, 50%, 75% of the physical memory), and TMM using its histogram-based limiting method. For vanilla Linux, we present results with and without the contributory application. The three different sets of applications represent different user activities with different working set sizes: Small (Mozilla), Medium (Mozilla, OpenOffice, and KView), and Large (Gnuplot with very large data set). We track the average and maximum number of page misses per second recorded in the first minute after the user comes back from break and present the results in Figures 2 and 3. Note that we rebooted between each trial, and we only monitored page misses incurred by opaque processes.

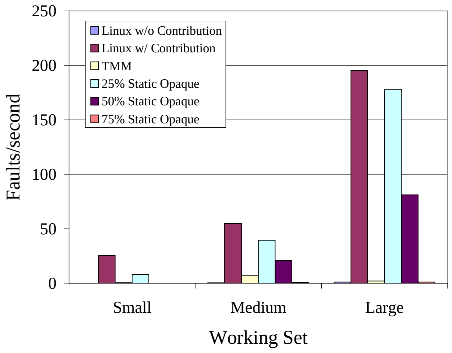

Figure 2: This figure shows the average page faults/sec in the first minute after resuming work. TMM performs much better than an unmodified system, and better than static limits, except for a very high static limit.

The first thing to note is that with a vanilla Linux kernel, the system running the contributory application performs very poorly, incurring as many as 190 page faults/sec on average. Assuming that the application is page fault limited, and given an average miss latency of 2.5ms, these faults cause a 50% slow down in the execution of the application. Qualitatively, we have observed that this is highly disruptive to the user. Second,

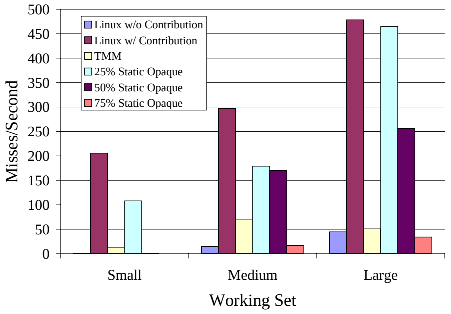

Figure 3: This figure shows the maximum page faults/sec in the first minute after resuming work. Short term violations of the target 5% slowdown are possible, but TMM performs better than unmodified and static systems.

for each static limit, there is a workload that performs worse than TMM, except for the 75% limit. In that case, the static limit performs well. However, we could have easily constructed a working set size larger than 75%, and TMM would have produced far fewer page misses than the static allocations.

Most importantly, the performance of TMM is comparable to the performance of the vanilla Linux system without contribution. The largest number of average page faults that TMM incurs is for the medium size working set, at 6.8 page faults/sec. By the same calculation used above, this faulting rate causes a 1.7% slow down in the opaque applications, well within our goal of 5%. This demonstrates TMM's ability to donate memory transparently. The few pages that TMM does evict could be recovered by making Linux's page eviction more LRU-like. As shown in Figure 3, short-term violations of our goal are possible—TMM statistically guarantees average performance over long periods. Nonetheless, TMM provides better short-term performance than the static limits (except 75%), and much better performance than unmodified Linux. For the large working set, even Linux without contribution does incur some page faults.

To measure the actual amount of memory that TMM donates to transparent applications, we ran the interactive phase of our benchmarks, waited for the limit to stabilize, and recorded the transparent limits. We also recorded the amount of memory donated under the static limits. The static limits apply to the limit on opaque memory and thus transparent memory contribution also depends on consumption by the operating system, thus a single static limit donates slightly different amounts of memory under different workloads. The results are presented in Figure 4.

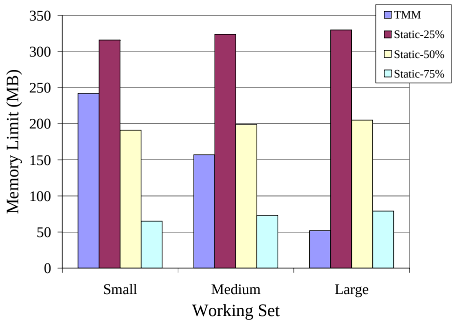

Figure 4: This figure shows the amount of memory in MB donated to transparent processes.

When comparing the amount of memory donated by each system, we show that TMM is conservative in the amount of memory it donates, favoring opaque page performance over producing a tight limit. Nonetheless, by considering both Figures 2 and 4, the results show that there is a working set for which the static limits fail to both preserve performance and contribute the maximum amount of memory. TMM succeeds in achieving both of these goals for every working set.

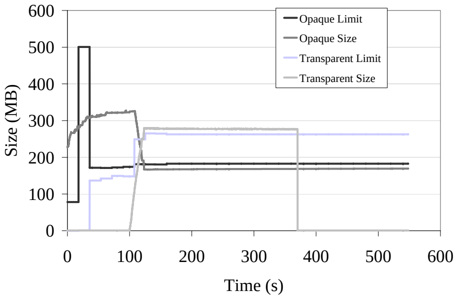

Figure 5: This figure shows a sample timeline of limits and utilization of the TMM system.

The next experiment demonstrates how TMM behaves over time. We conduct an experiment similar to the previous one and graph the memory use and memory limits that TMM sets. A timeline is shown in Figure 5. At the beginning of the timeline, the set of opaque processes is using 320 MB of memory and there are no transparent applications in the system. TMM has set a limit for both opaque and transparent processes, but as there is no memory pressure in the system, the memory manager lets opaque processes use more memory than the limit. Note that at this time, the sum of the transparent and opaque limits is far less than the physical memory of the machine (512 MB), the rest of the memory is in free pages. At 30 seconds we start a transparent process that quickly consumes a large amount of the free memory. TMM now sees a larger pool to divide and increases the transparent limit. TMM does not adjust the opaque limit as the user has not changed behavior and does not need any more memory than the limit allows. The transparent process is now causing memory pressure in the system, forcing pages out. The number of opaque pages is over its limit so it loses them to the evictor. The graph exhibits some steady state error. We have tracked this to the zoned memory system that Linux uses, something we will correct in a redesigned evictor.

Lastly, it is important to note the CPU and memory overhead of TMM. TMM consumes approximately 6% of the CPU during startup and less than 2% once it has established an accurate LRU queue. This CPU time is primarily due to running TMM in user space requiring many user-kernel crossings to exchange page references and limits. TMM uses approximately 64MB of memory. This large memory overhead is due to duplicating the kernel's LRU list in user-space, another straightforward optimization. A kernel implementation of TMM will reduce both of these overheads significantly.

5 Related Work

The CacheCOW system [6] generally defines the problem of providing QoS in a buffer cache. CacheCOW contains some elements of TMM, including providing different hit rates to different classes of applications, but targets Internet servers in a theoretic and simulation context. CacheCOW does not address many of the practical issues in using, gathering, and aging LRU histograms.

Resource Containers provide static allocations to resources such as physical memory [5] to prevent interference between different classes of applications. However, such static allocations are inherently ineffective and do not determine what to set the allocations to. We have shown in our evaluation the benefits of using a dynamic scheme, such as TMM, over static allocations. It is possible to adjust these limits at runtime, and therefore would be possible to use the memory tracing and LRU analysis techniques of TMM to adjust resource container limits, or to dynamically adjust other systems which control memory use.

LRU histograms [12], or page recency-reference graphs [11, 9], are useful in many contexts such as memory allocation and virtual memory compression and are essential to TMM. Some optimizations to gathering histograms have been implemented that rely on page protection but lower the overhead to 7-10% [13]. In this paper, we use an adapted form of tracing that avoids the overhead of handling relatively expensive page faults [2].

6 Conclusions

In this paper we present an OS mechanism, the Transparent Memory Manager (TMM), for supporting transparent contribution of memory. This system prevents contributory applications from interfering with the performance of a user's applications, while maximizing the benefits of harnessing idle resources. TMM protects the user's pages from unwarranted eviction and limits the impact on the performance of user applications to less than 5%, while donating hundreds of megabytes of idle memory.

Notes

[1] Downloadable from: *http://prisms.cs.umass.edu/software.html*

References

[1] G. Almasi, C. Cascaval, and D. A. Padua. Calculating stack distances efficiently. In *The International Symposium on Memory Management (ISMM 2002)*, Berlin, Germany, June 2002.

[2] J. Cipar, M. D. Corner, and E. D. Berger. Transparent contribution of storage and memory. Technical Report 06-05, University of Massachusetts-Amherst, Amerst, MA, January 2006.

[3] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36(SI):285–298, 2002.

[4] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[5] P. Druschel G. Banga and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, February 1999.

[6] P. Goyal, D. Jadav, D. Modha, and R. Tewari. Cachecow: Qos for storage system caches. In *International Workshop on Quality of Service (IWQoS)*, Monterey, CA, June 2003.

[7] Theodore Johnson and Dennis Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 439–450, Santiago, Chile, 1994.

[8] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th SOSP (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[9] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2), July 2003.

[10] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, November 1994.

[11] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of The 1999 USENIX Annual Technical Conference*, pages 101–116, Monterey, CA, June 1999.

[12] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the Third International Symposium on Memory Management (ISMM)*, Vancouver, October 2004.

[13] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamically tracking miss-ratio-curve for memory management. In *The Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, October 2004.