Virtual Memory-Induced Priority Inversion in Multi-Tasked Systems*

Gregory S. Hartman and Priya Narasimhan
Institute for Software Research International
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh. PA 15213-3890
{gghartma, priya}@cs.cmu.edu

Abstract

Virtual memory (VM) sub-systems in many widely adopted desktop and server operating systems rely on approximations of the least-recently-used (LRU) heuristic to select pages for replacement. These heuristics work well when memory is abundant, but they produce counter-intuitive behavior when applications' memory demands substantially exceeds the available physical memory. This paper describes the results of preliminary experiments with a new instrumentation framework that observes Linux VM behavior in a controlled setting. Repeated experiments with a micro-benchmark consistently reveal three types of misbehavior. First, the CPU scheduler's intended priorities can be inverted for an indefinite period of time when low-priority processes push higher-priority processes out of memory. Second, the VM heuristics can perpetually assign unequal amounts of memory to simultaneously running, identical processes. Finally, processes with modest memory requirements experience execution delays during periods of memory shortage.

Keywords: instrumentation, interactive, latency, LRU, memory shortage, page replacement, priority inversion, responsiveness, unfairness, virtual memory

1 Introduction

Virtual memory [3] is a well-known source of performance problems in computing systems. Designers of real-time systems typically choose operating systems without virtual memory support [7], or lock their applications into memory to avoid the potentially unbounded delays introduced by page faults [1]. However, eliminating virtual memory support is not an option for general software development, since doing so would force programmers to predict, precisely and ahead of run-time, the amount of memory needed by their software. In particular, web servers use software where the demand for memory is extremely difficult to predict. These systems use virtual memory to provide best-effort performance, given the available physical memory. When sufficient memory is not available, these systems are subject to performance problems, including thrashing, uneven partitioning [10], and lack of performance isolation. The first author of this paper confronted the following problems while supporting a high-volume commercial web site:

1. The failure of a single process running on the web server machine could make the entire web server unresponsive. At times, the performance degradation was so severe that it was difficult to gather the data needed to isolate the cause of the failure.

2. Estimating a process' demand for memory was difficult. The operating system provided no accurate indicator of the demand and, therefore, programmers were forced to estimate their applications' memory demand using knowledge of the application. This increased the risk of failures after upgrades; software which was technically correct occasionally failed to meet the performance requirements of the web site.

3. The data that the web servers used to process requests was created through a lengthy, resource intensive conversion process. The technicians who monitored this process observed that the memory partitioning on the systems they used to convert the data was difficult to predict. To meet their schedules reliably, the technicians were forced to run no more than a single conversion at any given time on a system. In addition to forcing the purchase of expensive hardware, this technique increased the burden on system administrators,

*The research work reported in this paper has been sponsored in part by the Army Research Office under Grant No. DAAD19-01-1-0646.

made the job of the technicians more confusing, and caused congestion on the network, which had to cope with large data transfers between the systems.

The complexity of the software on the web site made it difficult to reproduce these problems reliably and to distinguish between effects caused by the application code and the operating system.

We wanted to understand the operating system's contribution to these problems, and we wanted to be certain that our results were reproducible. Therefore, we developed an instrumentation framework and an automated testing system to observe the behavior of a trivial micro-benchmark. This paper describes the design of our instrumentation framework, and the statistics that we gathered. We also describe the kind of virtual memory imbalances that we encountered, and the conclusions that we drew from our observations and results. This paper demonstrates that simple micro-benchmarks can produce virtual memory misbehavior that is similar to the misbehavior observed with real-world applications.

2 Statistics of Interest

The operating system statistics of interest to our experiments are those related to virtual memory behavior, and are listed in this section. Our instrumentation framework examines the cumulative statistics given in Table 1 once every second in order to extract the following interval statistics for each running application on the system:

- *Resident set* (denoted by `pages` on the graphs) gives the number of the application's pages that are currently in memory.
- *Zero-on-write* is an optimization that allows the operating system to defer erasing newly allocated memory. Instead, the operating system maps an erased page multiple times into the application's address space with read-only permissions. On the first write, the operating system intercepts the minor fault, allocates a new page, and maps it as a read/write page. The `zow` field on the graphs indicates the number of page faults associated with the zero-on-write optimization.
- *Percentage CPU:* (denoted by `% CPU` on the graphs) gives the number of timer interrupts that occurred in the application. The timer on Linux generates an interrupt at a frequency of 100Hz; the monitor reads the statistics once per second, so the number of timer interrupts incurred by the application, within the monitoring interval, can be directly interpreted as the % CPU load. However, execution delays in the monitor can cause this statistic to exceed 100% for an application because timer interrupts will continue to accrue during the delay.
- *Major faults:* (denoted by `majflt` on the graphs) occur when the application attempts to access a page that is not in memory. The operating system suspends the application, and initiates a read for the page from the disk.
- *Minor faults:* (denoted by `minflt` on the graphs) occur when the application attempts to access a page and the page is discovered in memory. The zero-on-write optimization can cause this to happen. Minor faults also occur when the application accesses a page on the operating system's free list before it is reallocated.

In addition, the monitor extracts the following interval statistics by querying the global cumulative statistics given in Table 2:

- *Swap-in:* counts the number of pages read from the disk for all running applications.
- *Swap-out:* counts the number of pages written to disk for all running applications.
- *Jitter:* measures the variation in the time between the monitor's samples. Jitter is normalized so that it is zero in the expected case. Our monitor process uses an interval timer to avoid accumulating the delays, so small values of positive jitter are often followed by a corresponding negative value.

3 Instrumentation Framework

3.1 Design Objectives

Our objectives in the design of the instrumentation framework included:

- *Constant monitoring:* The monitor should minimize its use of system resources so that our framework can run continuously, even on production systems without creating performance problems of its own. Constant monitoring of a production system has two advantages. First, it allows us to gather data for intermittent performance problems which may be difficult to reproduce on a development system. Second, the monitor could be extended to alert system administrators to performance problems in a timely manner, ultimately reducing the time needed to correct the problem.
- *Reflective monitoring:* To be comprehensive, we required that our monitor process keep statistics for itself. This allows us to keep track of the monitor's own

resource consumption. The data that the monitor reads is updated by the operating system independently of the monitor process. If the monitor is blocked (for reasons such as page faults), there will be artifacts in the reported data. By tracking the behavior of the monitor, we can locate these artifacts, and exclude them from our analysis.

3.2 Assumptions

We made the following assumptions while designing our framework and while running our tests:

- The system always enters the same state after a reboot. Therefore, identical tests started at the same time after a reboot will produce identical results. We test this assumption by running the same test many times, and by comparing the results. Where the results differ, we document the differences.
- The monitor process does not affect the behavior of the system to the extent that our findings cannot be applied to systems that are not running the monitor. We have been able to reproduce the behavior described in this document even without the use of the monitor.
- The entries in the `/proc` file-system [8] that we use for gathering the kernel statistics accurately reflect the state of the system, as documented in the Linux man pages. We have examined the Linux kernel source code for the statistics of interest to us, and verified that the `/proc` implementation corresponds to its documentation.
- Linux has not been designed to favor processes based on their starting time. Therefore, unequal resource allocations to multiple, identical simultaneously executing applications constitutes undesirable behavior.

3.3 Implementation Details

Our framework runs over a Red Hat Linux 8.0 installation. We replaced the kernel with an enhanced stock (non-Red Hat) 2.4.20 kernel that provides additional statistics that allow us to observe the behavior of the page-reclamation algorithms. The test system has a Pentium 4 processor running at 2GHz with 512MB RDRAM memory and a 40GB IDE hard disk. Our instrumentation framework, consisting of a monitor, a test script and some kernel-level instrumentation, attempts to pinpoint virtual memory imbalances through a micro-benchmark that we designed.

Micro-benchmark. Our micro-benchmark is a simple memory scanner. It allocates a single 384MB buffer using the `malloc` routine in the C runtime library, and then writes sequentially to the entire buffer one byte at a time. When it gets to the end of the buffer, it returns to the beginning of the buffer, and continues to write sequentially. Instances of the micro-benchmark are called scanners in the rest of this paper. One scanner will clearly fit into the physical memory (512 MB) of our system; additional scanners will undoubtedly cause paging. This scanner was originally intended to validate memory allocation in order to test the statistics gathered by our monitor. Although the our current micro-benchmark provides valuable insights, we intend to apply our framework to more realistic workloads and applications in the future. We implemented the scanner on Linux, and have additionally ported it to FreeBSD and Windows XP.

Monitor. The monitor is a process that queries the `/proc` file-system once every second to extract the kernel statistics for every running process, and writes the values to a log file. The monitor gathers global statistics, such as the number of free pages, as well as process-specific statistics, such as the number of resident pages. The monitor currently uses the Linux `/proc` file-system to translate internal kernel data structures into a text representation. The per-process and global statistics of interest to us are listed in Tables 1 and 2, respectively. We note that the monitor does not gather the information in an atomic operation, so discrepancies can appear in the data. The monitor records timestamps from the CPU clock register to allow us to detect jitter between the samples. It also records, in the log, the first appearance of a process, and also a process's exit from the system.

The monitor also launches scanners. The first scanner is launched 12s after the monitor starts. After the the monitor launches a scanner, it waits for a fixed time interval before launching an additional scanner. The delay between launches and the number of scanners to launch are randomly selected at the beginning of each test-run. After running for one hour, the monitor terminates the test-run by rebooting the system.

Test Script. This script runs during the boot sequence on Linux. It halts the boot process, and launches the monitor before the networking sub-system starts, thereby minimizing the number of processes on the system, and also eliminating any network traffic that might perturb our experiments. The monitor and this script constitute a fully automated test system, greatly reducing the variability of the timing of the tests. The other processes running in the system at the time of the test include: `init`, `keventd`, `kapmd`, `ksoftirqd_CPU0`, `kswapd`, `bdflush`, `kupdated`, `mdrecoveryd`, `kjournald`, `rc`, `minilogd`, `initlog`, and `bash`. The `bash` process runs on the console, allowing the test to be terminated so that results can be collected from the system.

Label Name	Description
comm	Program name
ppid	UNIX process identifier of the parent process
minflt	Number of the page faults that didn't result in a disk I/O
majflt	Number of page faults that resulted in disk I/O
utime	Time spent in user-level code for this process
stime	Time spent in the operating system for this process
priority	Process priority for the Linux scheduler
rss	Number of pages in the page table of the process
zerofilled	Number of copy-on-write operations of the process
reclaimed	Number of pages reclaimed from this process
reclaimscans	Number of times that the kernel attempted to reclaim pages from the process

Table 1. Per-process statistics of interest. The monitor process logs a copy of these kernel counters for each task on the system. The `zerofilled`, `reclaimed`, and `reclaimscans` counters are our enhancements to the kernel to understand the behavior of the Linux page-swapping system.

Label Name	Description
memtotal	Total amount of memory in system, as reported from `/proc/meminfo`. (in kB).
memfree	Free memory, as reported from `/proc/meminfo`. (in kB).
memshared	Shared memory, as reported from `/proc/meminfo`. (in kB).
swapin	Number of pages read from swap files.
swapout	Number of pages written to swap files.
cycles	Number of processor cycles (w.r.t. our 2GHz testbed) since the last read.
readtime	Number of cycles spent reading the data from the `/proc` file-system.
fairness	Value of the fairness heuristic of any memory scanners; -1 if no scanners, 0 if completely fair, 1 if completely unfair.

Table 2. Global statistics of interest. This is a partial list of the counters that track the behavior of all the processes on the system. The statistics in bold are generated by the monitor.

Kernel Instrumentation We enhanced the Linux kernel with counters to track the behavior of the virtual memory sub-system as it allocates and reclaims pages. These additional counters are `zerofilled`, `reclaimed`, `reclaimscans`, and are described in more detail in Table 1. Before we added these statistics, the kernel did not provide any indication that memory had been reclaimed from an application.

4 Empirical Results

Using our framework, we have observed three behaviors that make virtual memory performance difficult to predict:

1. *Priority inversion* - the virtual memory sub-system does not respect the CPU scheduler's process priorities. Specifically, a low-priority process can steal pages allocated to a higher-priority process. In certain circumstances, the low-priority process might retain these pages indefinitely, suspending the higher-priority process when the latter encounters page faults. As a result, the lower-priority process obtains virtually exclusive CPU access, leading to priority inversion.

2. *Virtual memory imbalances* - identical applications obtain significantly different allocations of the available physical memory when they are run together. These imbalances exist in the operating systems that we have tested (Linux, FreeBSD, and Windows XP), and appear to be an artifact of the victim-selection algorithm. Our detection of virtual memory imbalance hinges on (i) our knowledge of our micro-benchmark's memory-related behavior, (ii) our use of three *identical,* simultaneously executing instances of our micro-benchmark, and (iii) our ability to observe unfairness in physical memory allocations across these instances.

3. *Execution delays* - processes may exhibit pauses on the order of one second when memory is scarce. This is true even of processes that run periodically, and that have small, predictable page-reference strings. The monitor process that we use to examine the Linux kernel counters for the purpose of tracking resource allocations on Linux was originally susceptible to this problem. We were able to work around this problem by locking the monitor into memory. While this approach does work for other applications, it may be infeasible for applications which use a large amount of memory, or those that process a mixture of high-priority and low-priority requests.

These problems stem from the page-replacement algorithm's ability to affect which applications experience page faults on multi-tasking systems. This, in turn, affects task scheduling, which ultimately affects the global order of page references in the system. When the page-replacement algorithm is driven from this global order, as is the case in the least-recently-used (LRU) heuristic [2] used by the virtual memory sub-systems in Linux [5, 6] and FreeBSD, a feedback loop can occur. While these effects have been known for some time [10], many operating system texts (for example [9]) present an oversimplified view of the virtual memory sub-system which does not address this problem. Unfortunately, the LRU heuristic penalizes applications with conservative memory and CPU usage in favor of applications that allocate memory and use the CPU liberally.

Our instrumentation framework also indicates that the existing kernel statistics do not provide enough information to quantify either an application's memory demands or the amount of free memory on the system. We demonstrate that the statistics in the Linux kernel can either over-estimate or under-estimate an application's demand for memory, depending on the available memory in the system at the time. To make matters worse, the accuracy of the statistics seems to improve only when memory is reclaimed from applications, which often results in execution delays.

We work around this limitation in our current research by observing multiple instances of a micro-benchmark which has a static, pre-specified demand for memory. Any difference in the runtime allocation of memory across the identical instances, therefore, serves as an indicator of imbalance in memory allocation. Real-world applications are not so simple; their memory demands may vary based on the stage of processing (for batch applications) and the input (for interactive applications). In addition, few systems are exclusively dedicated to running multiple copies of the same application. Therefore, as a part of our next research steps, we intend to find a better way of characterizing an application's demand for memory. Our conjecture is that we can enhance the kernel, with minimal impact on application performance, to observe the working-set [4] of applications, and that this working-set information will accurately reflect the memory demand of applications.

4.1 Priority Inversion and VM Imbalance

We describe a single test-run involving three scanners, each of which is started at different times (12s, 127s, and 242s, respectively). This test is representative of 81% of the test-runs using three scanners (*i.e.*, 36 out of 44 test-runs) that we have analyzed. The remaining 19% of the test-runs (called non-conforming tests) do not exhibit the unbalanced memory behavior; instead, the three scanners end up sharing the memory equally. We believe that the non-conforming tests are different because the system does not reach a stable state between the launches of the micro-benchmark; all of the non-conforming tests have a delay of less than 30s between micro-benchmark launches (as compared to the 115s inter-scanner interval described below).

The variation of the kernel statistics with the launch of the three scanners is shown in the graph in Figure 1.

- **Initial system behavior**: The first part of the graph shows the behavior of the system at the beginning of the test, when the system has just rebooted. The testing process pauses for 12s to ensure that the boot has completed. During this time, a large number of pages are free.

- **First scanner starts:** After these 12s have elapsed, the testing process launches the first scanner. A large number of minor faults occur between 12s and 14s; all of these seem to be caused by the zero-on-write optimization. Note that the process causes 75 major faults when it starts. Since there is plenty of memory available, the minor and major faults do not continue as the process runs. In addition, there is no swapping activity. The CPU load quickly approaches 100%, because the scanner never sleeps, and never has to wait for memory. The system reaches a stable state at 15s, and remains in this state until the second scanner starts.

- **Second scanner starts:** The monitor launches the second scanner at 127s. This scanner also causes 75 major faults when it starts; it also causes a large number of minor faults, most of which are associated with the zero-on-write optimization. Unlike at the launch of the first scanner, memory is now constrained. The second scanner's demand for new pages causes pages to be swapped out for the first time during the tests. In addition, the minor faults are spread over a 10s period instead of being clustered in a 3s window as they when the first scanner started. At 136s, the system begins to page in some of the memory that it swapped out. As a result, the CPU utilization falls below 5% once we are 144 seconds into the test-run.

- **Continued instability:** The system does not reach a stable point after the second scanner runs. In fact, the first scanner gains control of the memory and the CPU at 173s, only to lose it again at 203s! This event, however, is not present in all of the tests-runs using three scanners.

- **Third scanner starts:** The third scanner begins to run at 242s. Since there are few free pages at this time, its zero-on-write faults extend until 253s. There is a corresponding increase in swap-out activity as the operat-

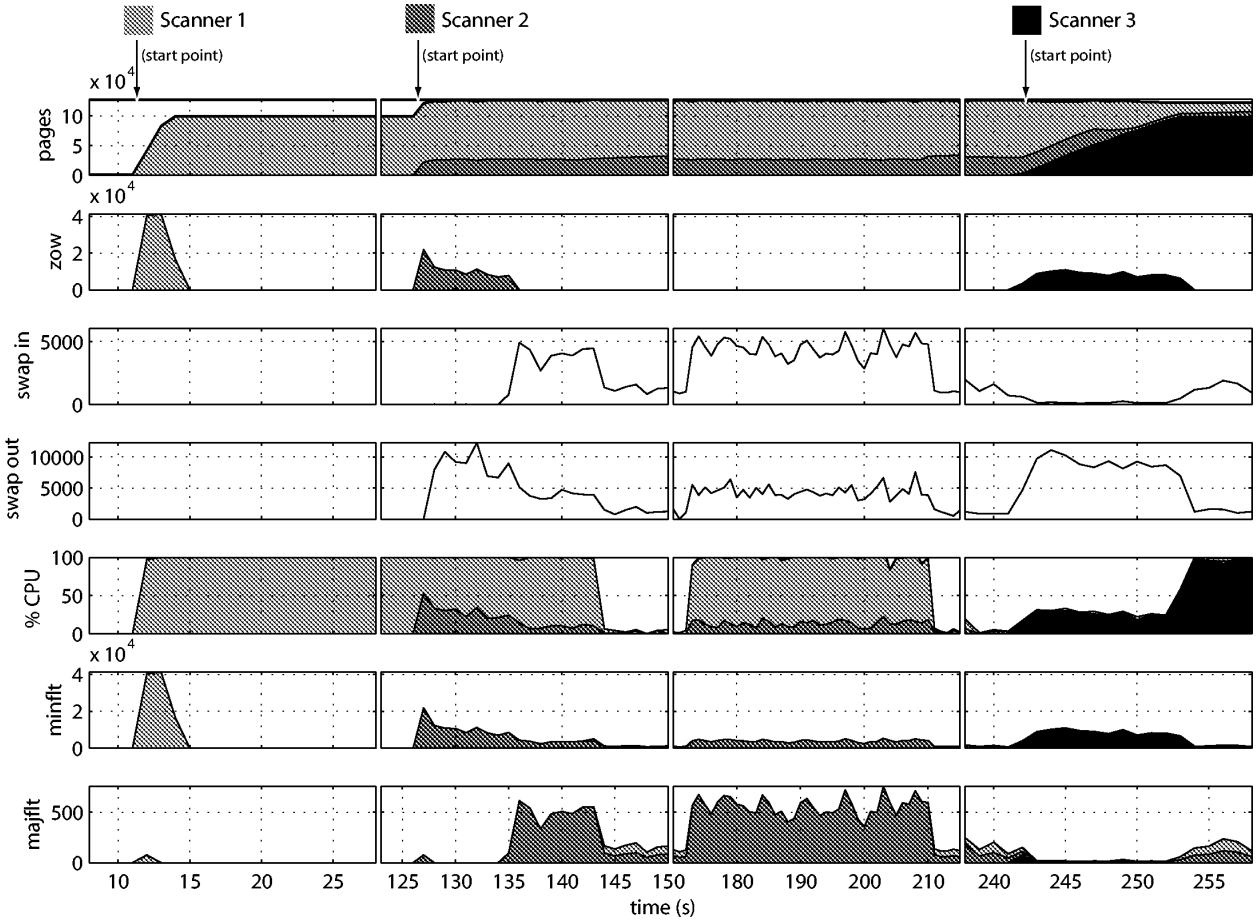

Figure 1. VM imbalance in a typical three scanner test. Column 1 shows the first scanner's launch; column 2 shows the second scanner's launch; column 3 shows a brief period of dominance for the first scanner; column 4 shows the third scanner's launch. The third scanner quickly pushes the others from memory.

ing system writes pages to disk. The new scanner consumes 20-30% of the CPU during this period of time. The third scanner's share of the memory gradually increases, until its all of its pages are in memory at 254s. At this point, the third scanner consumes nearly 100% of the CPU. While this increase occurs, there are very few swap-ins, probably indicating that the disk is saturated with the swap-out requests. The zero-on-write optimization gives the third scanner an advantage over the existing scanners; it can claim any page on the free list, zero it, and use it immediately. The first two scanners, on the other hand, must wait for data to be loaded from the disk before they can use new pages.

Once the system has reached this state, the third scanner retains its memory indefinitely, as seen in Figure 2. The third scanner can almost always run, since most of its pages are in memory. Therefore, its pages are not likely to be selected by the LRU heuristic. On the other hand, the first two scanners are almost always blocked as they wait for pages from the disk; many of their pages will, thus, be on the LRU list. This is because the LRU heuristic uses a global view of time rather than the execution times [4] of the individual scanners to age their respective pages. The problem may be addressed by employing the working-set heuristic to age a process's pages based on the process's execution time rather than on a global time-base.

The memory imbalance is so pronounced that it defeats the purpose of priorities in the Linux scheduler. This can be seen from the data in Table 3. This data indicates that at all of the sampling points beyond 260s (after the imbalance occurs and persists), the third scanner has a priority lower those of the first two scanners. This is expected behavior, because the third scanner is using nearly all of the

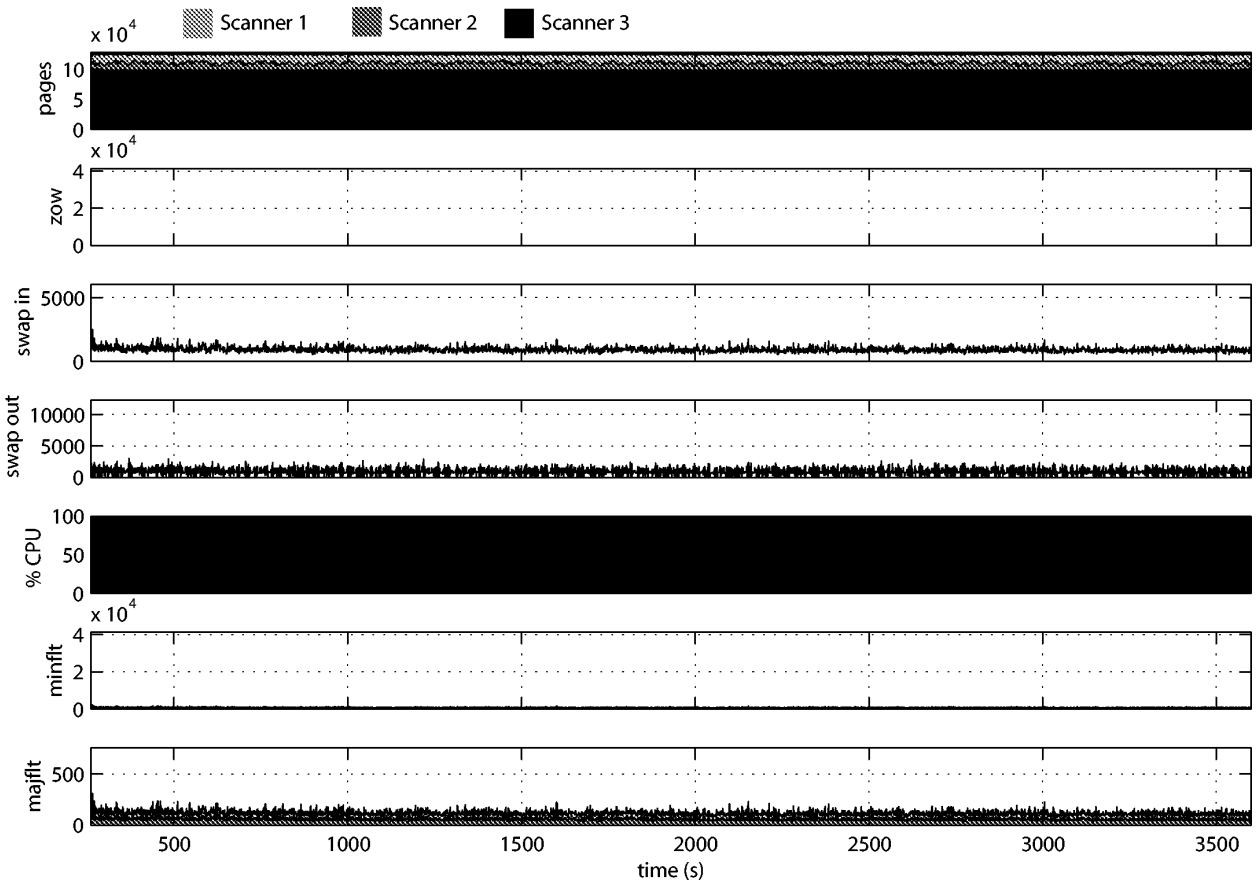

Figure 2. VM imbalance persists indefinitely.

	Min % CPU	Average % CPU	Max % CPU	Min Priority	Max Priority
1	0	1.6	8	9	12
2	0	1.6	8	9	11
3	84	96.1	107	14	20

Table 3. Priorities and CPU usages of the three scanners between 260s and 3600s of a representative test-run. On Linux, higher numeric priority values indicate lower-priority processes. The value of 107 in the Max % CPU column for scanner 3 is caused by a .07s irregularity in the scanner's sampling interval at some point during the test-run. Priority inversion occurs because the third scanner consistently runs at a lower priority than scanners 1 and 2, but gets a much larger share of the CPU.

CPU, and the default scheduler in Linux lowers the priority of such CPU-intensive processes. However, the behavior of the virtual memory system blocks the first two scanners. Therefore, they only consume only 1.6% of the CPU in spite of their status as high-priority processes, while the third scanner obtains 96.1% of the CPU. This is clearly a case of priority inversion.

This behavior does not emerge in all of the tests. In 19% of the test-runs (all of which had a delay between scanner launches of 30s or less), the memory between the three scanners was ultimately balanced. In a small number of tests, the memory spontaneously re-balances after being unbalanced for 20-40 minutes. We have not been able to reproduce this behavior consistently, and it seems to happen in approximately only 5% of the tests when the number of scanners and the delay between scanner launches are fixed. In addition, we can find nothing in our logged statistics to indicate the cause of this relatively infrequent stable behavior.

We ran some initial tests to verify that this behavior was not specific to the virtual memory implementation in the Linux kernel alone. Therefore, we ported the scanner to FreeBSD and Windows XP, and conducted our experiments. We were not able to port the monitor[1], so we ran the scanners manually, and observed the system behavior instead through the `top` utility (in the case of FreeBSD) and the Task Manager (in the case of Windows XP). The FreeBSD system exhibited the same instability that we observed on Linux. The observed behavior on Windows was somewhat different; when the second scanner starts, it is unable to recover memory from the first scanner. When the third scanner starts, the operating system drastically reduces the resident sizes of all of the scanners. This behavior may be desirable, since the memory does not reduce the faulting rate of the scanners. However, the GUI of Windows XP still exhibits large execution delays when three scanners are running. In our future work, we plan to quantify these delays and to log them in the monitor.

[1]The porting of the monitor was hindered because these operating systems do not provide the `/proc` interface for ready access of the kernel statistics.

4.2 Execution Delays

In our early experiments, we occasionally observed pauses of approximately 1.7s in the monitor output. These pauses seemed to occur just after the second scanner started. When we examined the logged data for the monitor process, we discovered that the monitor was experiencing major faults near these delays. This was somewhat surprising, because the monitor ran once every second, and "touched" the same addresses each time that it ran. On examining the data in the logs further, we concluded that the monitor needs to maintain 76 pages (304kB) of its pages in memory in order to avoid these major faults.

However, these pauses made the log files from the monitor difficult to interpret. Therefore, we decided to lock the monitor into memory, making its pages ineligible for replacement. This appears to confine the jitter to approximately 1/50th of a second, allowing us collect data even while the system is experiencing severe swapping. Figure 3 shows data from an early version of the monitor which experienced one of these pauses. In this test-run, the second scanner starts at 36s, causing the system to run low on memory. As a result, there are gaps in the data at 38s and 40s, (shown as blank spaces on the graph). At 40s, the monitor outputs a jitter value of 1.7s. This is consistent with the monitor missing two samples. The major fault data (graphed as `majflt`) shows that the monitor detected a total of 34 major faults at 39s and 41s. In this particular test-run, the monitor experiences one additional major fault. However, this fault does not seem to cause appreciable jitter in the output.

5 Predicting System Performance

The instrumentation of the virtual memory system does not allow us to make predictions about the system performance. To make reliable predictions, we would need to understand the memory demand of the applications. An application's memory demand is not equivalent to its resident size, as reported by the operating system. The working set[2] for an application may be much larger than what is indicated by the resident set in cases where the system is low on mem-

[2]"The working set is usually defined as a collection of recently referenced segments (or pages) of a program's virtual address space."[4]

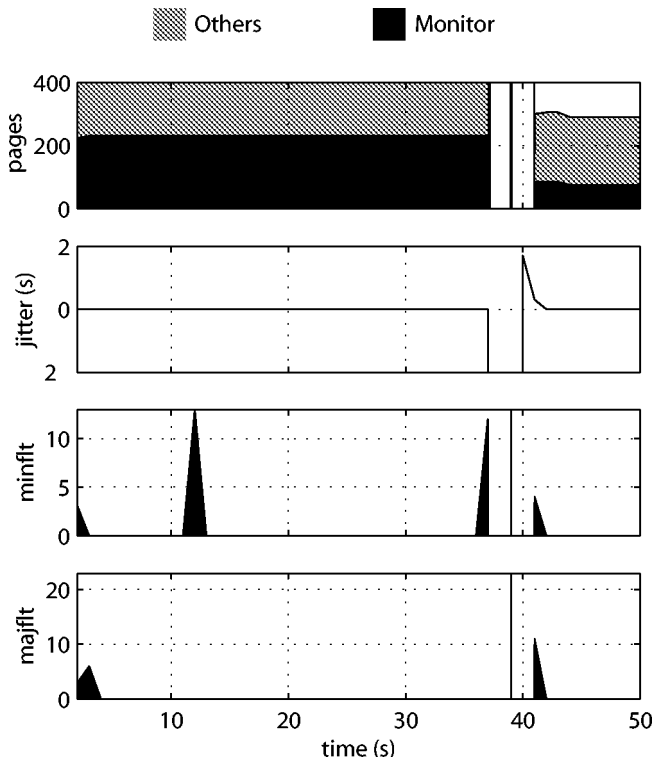

Figure 3. Jitter in the monitor process. The maximum of the y-axis on the *pages* graph has been lowered, and the non-monitor faults have been removed from the *majflt* and *minflt* graphs to make the monitor's statistics more apparent. The gaps at 38s and 40s are the result of missing log samples. Some data is available at 39s, and is represented as a thin vertical line. The 1.7s delay at sample 40 accounts for the two lost samples.

ory. This can be clearly seen in the fourth column on Figure 1. Here, the resident set accurately reflects the memory demand for the third scanner. However, the first two scanners have the same working set, but a much smaller resident set. Because these scanners generate a large number of major faults, we can conclude that the resident size is smaller than the current memory demand of the scanners. However, there do not exist kernel statistics that will tell us the difference between the current resident set and the true memory demand of the scanner. We believe that working sets represent one potential solution in order to provide a much more accurate estimate of the memory demand of applications.

On systems where the memory is not scarce, the resident set may overestimate the memory demand of an application. This can be seen on Figure 3. Here, the monitor's resident size starts out at 231 pages (see the data between 1s and 38s). However, when the number of free pages drops, the operating system begins to recover some of the pages. Once this recovery has finished, the resident size drops to 85 pages (between 41 and 43s). This is still larger than the actual memory demand since the resident size drops to 76 pages at 44s. This approach to quantifying an application's memory demand has several disadvantages:

- There is no way to know when the resident size accurately reflects the memory demand.
- The application often encounters major faults during the page reclamation, degrading its performance.
- The number of pages marked as free on the system is artificially low. This makes it difficult to predict the behavior of the system when new applications are run.

6 Future Work

We plan to continue our investigation into the relationship between the virtual memory subsystem and response times. The questions that we would like to address in future research include: What set of benchmarks reproduce the problems that realistic applications encounter on memory constrained systems? Does the size of the working-set accurately reflect an application's memory demand? Can we use this information to predict system behavior when applications are added to systems? Can we detect execution delays in unmodified applications without consuming a large portion of the system's resources? If so, how often do these delays occur, and what are causes of the delays?

7 Conclusion

In this paper, we have demonstrated that (i) it is relatively straightforward to reproduce virtual memory misbehaviors, such as priority inversion, (ii) these VM misbehaviors persist for a long period of time, (iii) identical applications can obtain radically different shares of the physical memory, depending on the state of the system when they are started, (iv) execution delays afflict all applications on the system, even those that are written to make careful use of memory, and (v) existing kernel statistics do not provide a clear picture of applications' memory demand and of the system's available memory.

Virtual memory is used on a wide array of systems, ranging from desktops to large servers. Given the information above, we conclude that applications on these systems are vulnerable to VM-induced performance failures. These failures could be triggered by unreasonable requests, buggy software, or malicious programs. In addition, these systems are difficult to configure—while a system's statistics might

indicate that it has insufficient memory, these statistics cannot be used to quantify the amount of additional memory needed to achieve acceptable application performance. Our future work will examine the behavior of virtual memory (and of other resources) in order to provide applications with a stable, predictable platform.

8 Acknowledgments

We would like to thank the anonymous reviewers for their comments that helped us to improve this paper. We also thank Tim Halloran and Jay Wylie for helping us to clarify our ideas in the final version of the paper.

References

[1] *Digital UNIX Guide To Realtime Programming*. Digital Equipment Corporation, March 1996.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, July 1966.

[3] P. J. Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[4] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, 1980.

[5] M. Gorman. Code commentary on the Linux virtual memory manager. Technical report, University of Limerick, 2003.

[6] M. Gorman. Understanding the Linux virtual memory manager. Technical report, University of Limerick, 2003.

[7] H. Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[8] A. Rubini and J. Corbet. *Linux Device Drivers*, pages 103–108. O'Reilly and Associates, Inc., 2nd edition, 2001.

[9] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 6th edition, 2003.

[10] A. J. Smith. Multiprogramming and memory contention. *Software–Practice and Experience*, 10(7):531–552, 1980.