SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:295–318 (DOI: 10.1002/spe.437)

SP&E

TPF: a dynamic system thrashing protection facility

Song Jiang and Xiaodong Zhang*,†

Department of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795, U.S.A.

SUMMARY

Operating system designers attempt to keep high CPU utilization by maintaining an optimal multiprogramming level (MPL). Although running more processes makes it less likely to leave the CPU idle, too many processes adversely incur serious memory competition, and even introduce thrashing, which eventually lowers CPU utilization. A common practice to address the problem is to lower the MPL with the aid of process swapping out/in operations. This approach is expensive and is only used when the system begins serious thrashing. The objective of our study is to provide highly responsive and cost-effective thrashing protection by adaptively conducting priority page replacement in a timely manner. We have designed a dynamic system Thrashing Protection Facility (TPF) in the system kernel. Once TPF detects system thrashing, one of the active processes will be identified for protection. The identified process will have a short period of privilege in which it does not contribute its least recently used (LRU) pages for removal so that the process can quickly establish its working set, improving the CPU utilization. With the support of TPF, thrashing can be eliminated in its early stage by adaptive page replacement, so that process swapping will be avoided or delayed until it is truly necessary. We have implemented TPF in a current and representative Linux kernel running on an Intel Pentium machine. Compared with the original Linux page replacement, we show that TPF consistently and significantly reduces page faults and the execution time of each individual job in several groups of interacting SPEC CPU2000 programs. We also show that TPF introduces little additional overhead to program executions, and its implementation in Linux (or Unix) systems is straightforward. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: Linux operating system; page replacement; thrashing protection; virtual memory

*Correspondence to: Xiaodong Zhang, Department of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795, U.S.A.
†E-mail: zhang@cs.wm.edu

Contract/grant sponsor: U.S. National Science Foundation; contract/grant number: CCR-9812187, EIA-9977030 and CCR-0098055
Contract/grant sponsor: Sun Microsystems; contract/grant number: EDUE-NAFO-980405

Received 12 April 2000
Revised 16 October 2001
Accepted 16 October 2001

1. INTRODUCTION

1.1. MPL versus system thrashing

Multiprogramming level, simplified as MPL, is defined as the number of active processes in a system. We refer to these active processes in an multiprogramming environment as *interacting processes*, because they are competing for CPU and memory resources interactively. Dynamically maintaining an optimal MPL to keep a high CPU utilization has been a fundamental issue in the design of operating systems [1]. Operating system designers aim at providing an optimal solution to the problem of using the CPU and memory resources effectively in multiprogramming, while avoiding the thrashing that multiprogramming can cause. CPU utilization can be increased by increasing MPL—running more processes. However, as MPL increases to a certain degree, the competition for memory pages among processes becomes serious, which can eventually cause system thrashing, and CPU utilization will then be significantly lowered. Considering large variations of memory demands from multiple processes and dynamical memory requirements in their lifetimes of the processes, it is not practically possible to set a pre-defined optimal MPL in order to avoid thrashing while allowing a sufficient number of processes in the system. Existing operating systems, such as BSD and Solaris, provide load control facility to swap out and in processes, if necessary, for thrashing protection. This facility allows the systems to adaptively lower MPL, but process swapping can be quite expensive for both systems and user programs.

1.2. Thrashing and page replacement

Thrashing events can be directly affected by how page replacement is conducted. Most operating systems adopt global LRU replacement to allocate the limited memory pages among competing processes according to their memory reference patterns. With an increase in MPL, memory allocation requests become more demanding. To keep more processes active, limited memory space should be fully utilized. The global LRU page replacement policy follows this principle. However, the effort to improve memory utilization could cause low CPU utilization.

In a multiprogramming environment, global LRU replacement selects an LRU page for replacement throughout the entire user memory space of the computer system. The risk of low CPU utilization increases if the memory page shortage happens throughout the interacting processes. For example, a process is not able to access its resident memory pages when the process is resolving page faults. These already obtained pages may soon become LRU pages when memory space is being demanded by other processes. When the process is ready to use these pages in its execution turn, these LRU pages may have already been replaced to satisfy memory requests of other processes. The process then has to request the virtual memory system to retrieve these pages by replacing LRU pages of others, significantly lowering the system throughput with just a small increase in MPL. The page replacement may become chaotic, and could cascade among the interacting processes, eventually causing system thrashing. Once all interacting processes are in the waiting queue due to page faults, the CPU is doing little useful work.

1.3. Effectiveness of adaptive page replacement

Existing operating systems protect thrashing at the process scheduling level by load controls. A commonly used mechanism is to suspend/reactivate or swap out/in programs to free more memory space after the thrashing is detected. For example, the 4.4 BSD operating system [2] initially suspends a program after thrashing. If the thrashing continues, additional programs are suspended until enough memory becomes available. Our experiments and analysis show that there are several system performance advantages for conducting adaptive page replacement over process scheduling for the purpose of thrashing protection. First, since improper page replacement during process interactions is a major and internal source of system thrashing, a solution to adaptively adjust page replacement behavior to current system needs can be fundamentally effective to address the problem. Second, the alternatives of load controls are limited to suspend or remove existing processes. Since this approach is expensive and can dramatically degrade user programs' interactivity, it is only used when the system is seriously thrashing. Finally, using the adaptive page replacement in the first place, we are able to eliminate the thrashing in its early stage, or significantly delay the usage of load controls. With adaptive page replacement and load controls guarding at two different levels and two different stages, the system performance will become more stable and cost-effective.

1.4. Our work

The objective of our study is to provide highly responsive and cost-effective thrashing protection by dynamically detecting and adaptively taking necessary actions at the kernel level of page replacement. It can also be regarded as page replacement adaptive to the system situation. We have designed a dynamic system Thrashing Protection Facility (TPF) in the system kernel considering the trade-off between CPU and memory utilizations. Once TPF detects system thrashing, one of the interacting processes will be identified for protection. The identified process will have a short period of privilege during which it does not contribute its LRU pages for removal. This allows the process to quickly establish its working set. With the support of TPF, early thrashing can be eliminated at the level of page replacement, so that process swapping will be avoided or delayed until it is truly necessary. TPF also improves system stability when memory is dynamically and competitively demanded by interacting processes. We take the Linux kernel as a case study to illustrate why TPF is needed and how it works.

2. EVOLUTION OF PAGE REPLACEMENT IN LINUX KERNEL

Linux, like most other systems, uses an approximate LRU scheme to keep the working set of a process in the system, and to contribute already allocated pages which may not be used in the near future to other interacting pages. A clock algorithm [3] is used, because it provides acceptable approximation of LRU, and it is cheap to implement, where NRU (not recently used) pages are selected for replacement.

Current page replacement implementation in Linux is based on the following framework. The interacting processes are arranged in an order to be searched for NRU pages when few free pages are available in the user space, and/or they are demanded by interacting processes. The system examines each possible process to see if it is a candidate from which NRU pages can be found for replacement. The kernel will then check through all of the virtual memory pages in the selected process. In a moderately loaded system, we could hardly observe execution performance differences due to the different page replacement implementations. However, when processes are competitively demanding memory allocations, interacting processes may chaotically replace pages among themselves, leading to thrashing. We take the three recent versions of Linux kernels to illustrate how the thrashing source is introduced from page replacement, and why a non-adaptive replacement policy has difficulty avoiding thrashing.

2.1. Kernel 2.0

In Kernel 2.0, the NRU page contributions are proportionally distributed among interacting processes. There is a ‘swap_cnt’ variable for each process, which is initialized with a quantity (RSS/1 MB) proportional to its resident set size (RSS). Once an NRU page is taken away from the process, its ‘swap_cnt’ will be decreased by one. Only when its ‘swap_cnt’ becomes zero, or the searching for an NRU page fails in resident space of the process, is the next process in the process list examined. When a process with a ‘swap_cnt’ of zero is encountered, it will be re-initialized using the same proportion rule. This strategy effectively balances memory usage by making all the processes provide proportional NRU pages. However, a major disadvantage of this approach is its high potential for thrashing, resulting in low CPU utilization. This is because when all the memory-intensive processes are struggling to build a working set under heavy memory loads, all are requesting more pages through page faults, and no one will be given a priority for the purpose of thrashing protection.

2.2. Kernel 2.2

In order to address the limit, Kernel 2.2 makes each identified process continuously contribute its NRU pages until no NRU pages are available in the process. Attempting to increase CPU utilization, this strategy allows the rest of the interacting processes to build up their working sets more easily by penalizing the memory usage of one process at a time. Below is the major section of code to select a process for page replacement in the kernel function ‘swap_out’ in mm/vmscan.c [4].

```
for (; counter >= 0; counter--) {
  max_cnt = 0;
  pbest = NULL;
select:
  read_lock(&tasklist_lock);
  p = init_task.next_task;
  for (; p != &init_task;
       p = p->next_task) {
    if (!p->swappable)
      continue;
    if (p->mm->rss <= 0)
      continue;
    /* Refresh swap_cnt? */
    if (assign == 1)
      p->mm->swap_cnt = p->mm->rss;
    if (p->mm->swap_cnt > max_cnt) {
      max_cnt = p->mm->swap_cnt;
      pbest = p;
    }
  }
  read_unlock(&tasklist_lock);
  if (assign == 1)
    assign = 2;
  if (!pbest) {
    if (!assign) {
      assign = 1;
      goto select;
    }
    goto out;
  }

  if (swap_out_process(pbest, gfp_mask))
    return 1;
}
out:
return 0;
```

In this section of code, the 'swap_cnt' variable for a process's data structure can be thought of as a 'shadow RSS', which becomes zero when a swap-out operation of a process fails. The 'swap_cnt's of all the swappable processes will be re-assigned with the respective RSS in the second pass through the process list in the inner loop when they all become zero. This inner loop will select the swappable process with the maximal RSS that has not yet been swapped out. Variable 'counter' is used to control how many processes are searched before finding an NRU page. We can see that once a process provides an NRU page, which means it is the one with the maximum 'swap_cnt' currently, the process will be selected for swapping upon the next request. This allows its NRU pages continuously to be replaced until a failure on finding an NRU page in the process occurs. Compared with previous kernel version, in addition to the changes in the selection of processes for NRU pages, there has been another major change in this kernel. In Kernel 2.0, there is an 'age' associated with each page, which is increased by three when it is referenced, called *page aging*, and decreased by three each time the page is examined. Once the 'age' decreases to zero, it becomes an NRU page and is ready to be replaced. Kernel 2.2 greatly simplifies the structure by eliminating the 'age' and only making use of the reference bit of each page in the PTE (page table entry). The bit is set when the page is referenced and reset when the page is examined. The pages with reference bits of zero are NRU pages and ready to be replaced. This implementation produces NRU pages more quickly for a process with a high page fault rate. These changes in Kernel 2.2 take a much more aggressive approach to make an examined process contribute its NRU pages, attempting to help other interacting processes to establish their working sets to fully utilize the CPU.

We have noted the effort made in Kernel 2.2 to retain CPU utilization by avoiding widely spreading page faults among all the interacting processes. However, such an effort increases the possibility of replacing fresh NRU pages in the process being examined, while some NRU pages in other interacting processes that have not been used for a long time continue to be kept in memory. This approach benefits CPU utilization at the cost of lowering memory utilization. Fortunately, in our experiments, we find that each interacting process is still examined periodically with a reasonable time interval. Although the average time interval in Kernel 2.2 is longer than that in Kernel 2.0, it seems to be sufficiently short to let most interacting processes have a chance to be examined. Thus memory utilization is not a major concern. However, the risk of system instability caused by low CPU utilization remains.

2.3. Kernel 2.4

The latest Linux kernel is version 2.4, which makes considerable changes in the paging strategy. Many of these changes focus on addressing concerns on memory performance that arose in Kernel 2.2. For example, without page aging, NRU replacement in Kernel 2.2 can not accurately distinguish the working set from incidentally accessed pages. Thus, Kernel 2.4 has to reintroduce page aging, just as Kernel 2.0 and FreeBSD do. However, page aging could help processes with high page fault rates to keep their working sets, causing other processes also to have high page fault rates to trigger thrashing. Kernel 2.4 distinguishes the pages with age of zero and those with positive ages by separating them into non-active and active lists, respectively, to prevent inefficient interactions between page aging and page flushing [5]. Unfortunately, this change does not help protect the system against thrashing, because the system still has no knowledge about which working sets of particular processes should be protected when frequent page replacement takes place under heavy memory workload. A similar argument can be applied in BSD and FreeBSD, where a system-wide list of pages forces all processes to compete for memory on an equal basis.

To make memory more efficiently utilized, Kernel 2.4 reintroduces the method used in Kernel 2.0 for selecting processes to contribute NRU pages. Going through a process list each time, it uses about 6% of the address space in each process to search NRU pages. Compared with Kernel 2.2, this method increases its possibility of thrashing.

2.4. CPU and memory utilizations rooted in page replacement

From the evolution of recent Linux kernels, we can see that in VM designs and implementations, finding an optimal MPL concerning thrashing has been translated into considerations of the tradeoff between the CPU and memory utilizations. For the purpose of high CPU utilization, we require that CPU not be idle when there are computing demands from ‘cycle-demanding’ processes. For the purpose of high memory utilization, we require that no idle pages be retained when there are memory demands from ‘memory-demanding’ processes. Our analysis has shown that the conflicting interests between the requirements on CPU and memory utilizations are inherent in a multiprogramming system. Regarding CPU utilization, the page replacement strategy should keep at least one process active in the process queue. Regarding memory utilization, the page replacement strategy should apply the LRU principle consistently to all the interacting processes. No process should hide its old NRU pages from swapping while other processes contribute their fresh NRU pages. It is practically difficult to design a strategy in favor of both CPU and memory utilizations with the low risk of system instability leading to thrashing. Current systems lack effective mechanisms to integrate the two requirements for the purpose of thrashing protection.

From the above code analysis, the page replacement implementation in Kernel 2.2 is more effective in thrashing protection than Kernel 2.0 and Kernel 2.4. However, we will show that the critical weakness resulting from the conflicting interests between the requirements on CPU and memory utilizations exists in Kernel 2.2. Thus, we implement our TPF in Kernel 2.2. The additional thrashing protection provided by TPF in this environment clearly demonstrates its effectiveness.

3. EVALUATION OF PAGE REPLACEMENT IN LINUX KERNELS 2.2

3.1. Experimental environment

Our performance evaluation is experimental measurement based. The machine we have used for all experiments is a Pentium II of 400 MHz with physical memory space of 384 MB. The operating system is Redhat Linux release 6.1 with the Kernel 2.2.14. Program memory space is allocated in units of 4 kB pages. The disk is an IBM Hercules with capacity of 8450 MB.

When memory related activities in program execution occur, such as memory accesses and page faults, the system kernel is heavily involved. To gain insight into VM behavior of application programs, we have monitored program execution at the kernel level and carefully added some instrumentation to the system. Our monitor program has two functions: user memory space adjustment and system data collection. In order to flexibly adjust available memory space for user programs in experiments, the monitor program can serve as a memory-adjustment process requesting a memory space of a fixed size, which is excluded from page replacement. The available user memory space can be flexibly adjusted by running the memory-adjustment process with different fixed sizes of memory demand. The difference between the physical memory space for users and the memory demand size of the memory-adjustment process is the available user space in our experiments.

In addition, the monitoring program dynamically collects the following memory system status quanta periodically for every second during execution of programs.

- *Memory allocation demand* (MAD) is the total amount of requested memory address space in pages. The memory allocation demand quantum is dynamically recorded in the kernel data structure of *task_struct*, and can be accurately collected without intrusive effect on program execution.
- *Resident set size* (RSS) is the total amount of physical memory used by a process in pages, and can be obtained from the kernel data structure of *task_struct*.
- *Number of page faults* (NPF) is the number of page faults of a process, and can be obtained from *task_struct* of the kernel. There are two types of page faults for each process: minor page faults and major page faults. A minor page fault will cause an operation to relink the page table to the requested page in physical memory. The system can mark minor pages as ‘not in address space’ without removing them from the main memory. In the time between the marking and the actual removal, a marked page may be reclaimed without a disk access. The timing cost of a minor page fault is trivial in the memory system. A major page fault happens when the requested page is not in memory and has to be fetched from disk. We only consider major page fault events for each process in this work, which can also be obtained from *task_struct*.
- *Number of accessed pages* (NAP) is the number of pages accessed by a process within a time interval of 1 s. During program execution, a system routine is periodically called to examine all the reference bits in the page table of a specified process. This quantum is only collected for dedicated executions of benchmark programs.

Table I. Execution performance and memory related data of the three benchmark programs.

Programs	Description	Input file	Memory requirement (MB)	Lifetime (s)
gcc	optimized C compiler	166.i	145.0	218.7
gzip	data compression	input.graphic	197.4	248.7
vortex	database	lendian1.raw	115.0	342.3
vortex	database	lendian3.raw	131.2	398.0

We have selected three memory-intensive application programs from SPEC CPU2000: *gcc*, *gzip*, and *vortex*. Using the system facilities described above, we first run each of the three programs in a dedicated environment to observe the memory access behavior without major page faults and page replacement (the demanded memory space is smaller than the available user space). Table I presents the basic experimental results of the three programs, where the 'description' column gives the application nature of each program, the 'input file' column is the input file names from the benchmarks, the 'memory requirement' column gives the maximum memory demand during the execution, and the 'lifetime' column is the execution time of each program. The 'lifetime' of each program is measured without memory status quanta collection involved. These numbers for each program represent the mean of five runs. The variation coefficients calculated by the ratio of the standard deviation to the mean is less than 0.01.

3.2. Page replacement behavior of Kernel 2.2

The memory usage patterns of the three programs are plotted by memory–time graphs. In the memory–time graph, the x axis represents the execution time sequence, and the y axis represents three memory usage curves: the MAD, the RSS, and the NAP. The memory usage curves of the three benchmark programs measured by MAD, RSS, and NAP are presented in Figures 1 (gcc), 2 (gzip), and 3 (vortex1, which is vortex with input file of 'lendian1.raw'). One major reason for the RSS of the gcc curve dropping is because the program itself frees its allocated memory space periodically for efficiency. We find that Linux Kernel 2.2.14 still provides a high potential for interacting processes to chaotically replace pages among themselves, significantly lowering CPU utilization and causing thrashing if the page replacement continues under heavy load. To show this, we have monitored executions and memory performance of several groups of multiple interacting programs. To make the presentation easily understandable on how memory pages are allocated among processes and their effects on CPU utilization, we only present the results of running two benchmark programs together as a group. We present three program interaction groups: gzip+vortex3 (vortex3 is vortex with the input file of 'lendian3.raw'), gcc+vortex3, and vortex1+vortex3. The available user memory space was adjusted by the memory-adjustment program accordingly so that each interacting program had considerable performance degradation due to 27–42% memory shortage. The shortage ratios are calculated based on the maximum memory requirements. In practice, the realistic memory shortage ratios are smaller due to dynamically changing memory requirements of interacting programs.

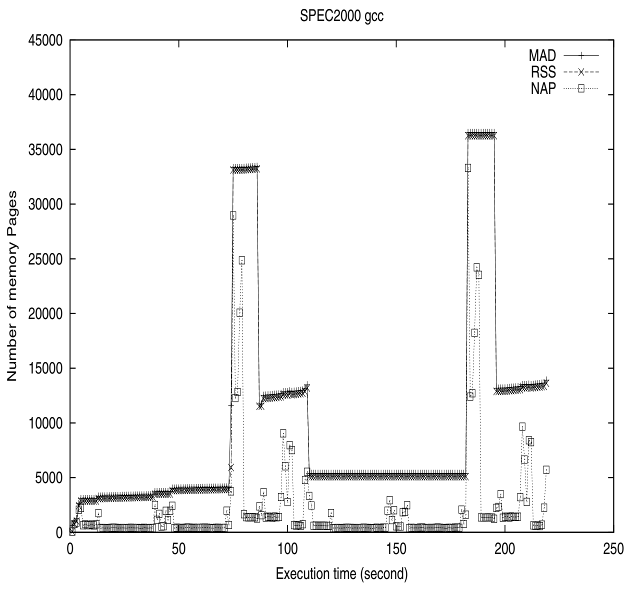


Figure 1. The memory performance of gcc in a dedicated environment.

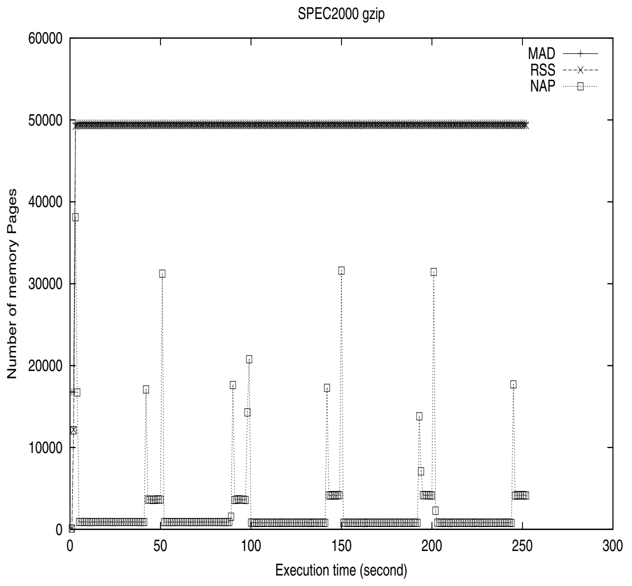


Figure 2. The memory performance of gzip in a dedicated environment.

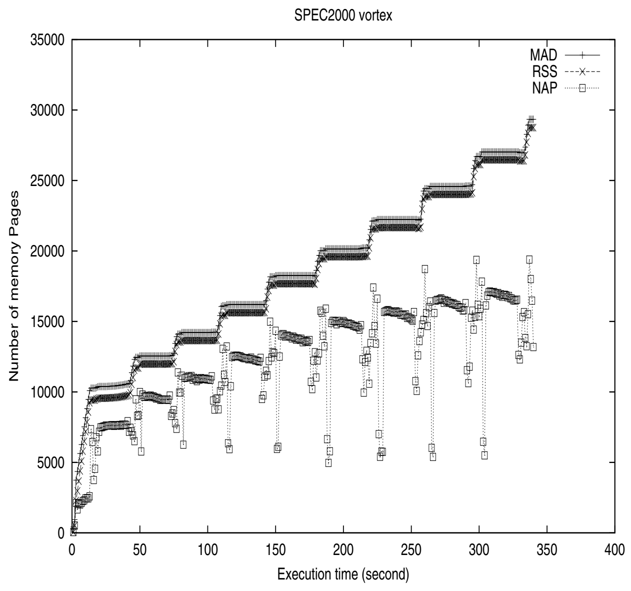


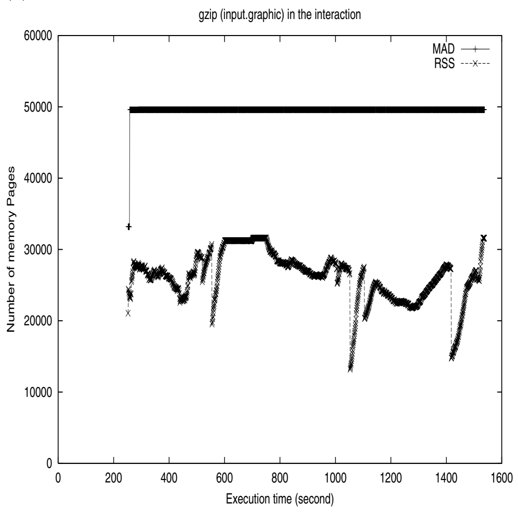
Figure 3. The memory performance of vortex1 in a dedicated environment.

Figures 4(a) and (b) present the memory usage behavior measured by MAD and RSS of interacting programs gzip and vortex3, respectively. After we added gzip to interact with vortex3 at 250 s, we observed that both their RSS curves are up and down in most of the times. CPU utilization is lower than 50% during the interaction because both processes were held in the waiting list by page faults for the most time. Adding more processes would worsen the case due to the lack of free memory in the system. We found that at around 620 and 780 s, gzip did get its working set and ran with a small number of page faults. Unfortunately, it went back to chaotic competition after that period. The measurement shows that the slowdown of gzip is 5.23, and that for vortex3 is 3.85.

Figures 5(a) and (b) present the memory usage behavior measured by MAD and RSS of interacting programs gcc and vortex3, respectively. For program vortex3, the RSS curve suddenly dropped to about 14 000 pages after it reached 26 870 pages, which was caused by the memory competition of the partner program gcc. After that, the RSS curves entered a fluctuating stage, causing a large number of page faults in each process to extend the first spike of gcc in the MAD and RSS curves to 865 s, and to extend a RSS stair in vortex to 563 s. In this case the slowdown of program gcc is 5.61, and that for vortex is 3.37.

Figures 6(a) and (b) present the memory usage behavior measured by MAD and RSS of interacting programs vortex1 and vortex3, respectively. Although the input files are different, the memory access patterns of the two programs are the same. Our experiments show that the RSS curves of both vortex programs changed similarly during the interactions. To favor memory utilization, NRU pages were allocated between the two processes back and forth, causing low CPU utilization and poor system

(a)



(b)

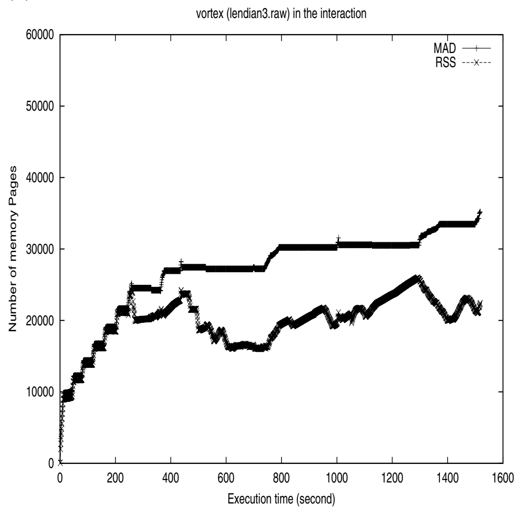
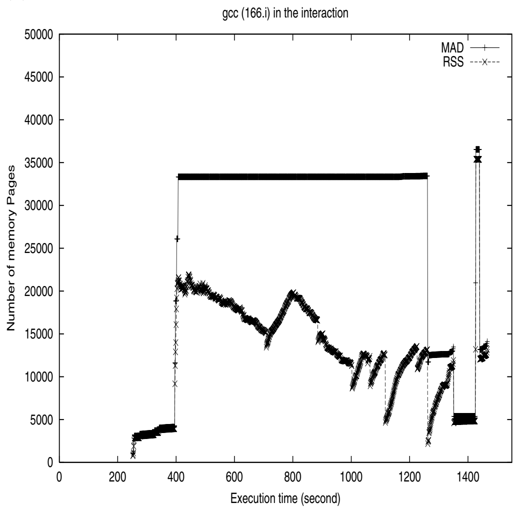


Figure 4. The memory performance of (a) gzip and (b) vortex3 during the interactions. Note: the execution of gzip started 250 s after that of the vortex3 program.

(a)



(b)

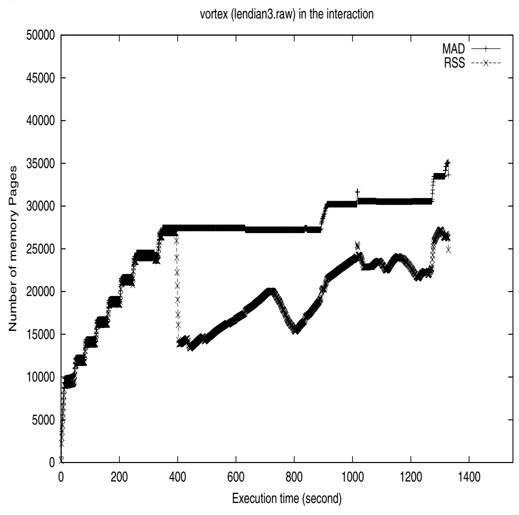


Figure 5. The memory performance of (a) gcc and (b) vortex3 during the interactions. Note: the execution of gcc started 250 s after that of the vortex3 program.

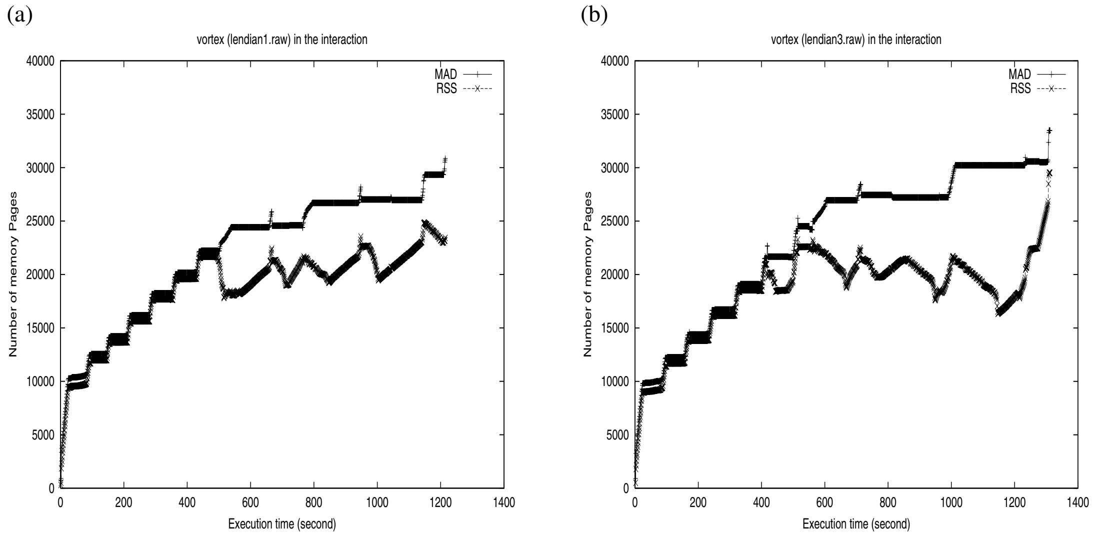


Figure 6. The memory performance of (a) vortex1 and (b) vortex3 during the interactions.

performance. After the RSS curves of both programs reached about 22 000 pages, their MADs could not be reached due to memory shortage. Our experiments again show that the execution times of both programs were significantly increased due to the page faults in the interaction. The slowdown for vortex1 is 3.58, and that for vortex3 is 3.33.

Our experiments indicate that although thrashing could be triggered by a brief, random peak in memory demand of a workload, the system may continue thrashing for an unacceptably prolonged time. To make a system more resilient against dynamically changing virtual memory load, a dynamical protection mechanism is desirable instead of a brute-force process stop, such as process suspension or even process removal.

4. THE DESIGN AND IMPLEMENTATION OF TPF

We propose to implement TPF as part of the page replacement for thrashing protection in order to improve system stability under a heavy load. The main idea of TPF is simple and intuitive. Once the system detects high page fault rates and low CPU utilization caused by multiple processes, TPF will identify a process and help it quickly establish its working set by temporarily granting a privilege to the process for its page replacement. With this action, the CPU utilization quickly increases because at least one process is able to do useful work. In addition, the memory space is expected to be released soon by the process after its completion, so that the memory demands of other processes can be satisfied.

We have implemented TPF in the Linux Kernel 2.2.14, which consists of two kernel utilities: detection and protection routines.

The detection routine is used to dynamically monitor the page fault rate of each process and the CPU utilization of the system. The protection routine will be awakened to conduct priority-based page replacement when CPU utilization is lower than a predetermined threshold, and when the page fault rates of more than one interacting process exceed a threshold. The protection routine then grants a privilege to an identified process among those with high page fault rates, which will only contribute a limited number of NRU pages. The identified process is the one that has the smallest difference between its MAD and its RSS (the least memory demanding process). The detection routine also monitors whether the identified process has lowered its page fault rate to a certain degree. If so, its privilege will be disabled. This action will retain memory utilization by treating each process equally.

4.1. The detection routine

There are four predetermined parameters in TPF:

1. CPU_Low is the lowest CPU utilization the system can tolerate;
2. CPU_High is the targeted CPU utilization for TPF to achieve;
3. PF_Low is the targeted page fault rate of the identified process for TPF to achieve; and
4. PF_High is the page fault rate threshold of a process to potentially cause thrashing.

We add one global linked list, high_PF_proc, in the kernel to record interacting processes with high page fault rates. Once we find the current page fault of a process exceeds PF_High, we will enter it in the linked list.

We have also added three new fields in *task_struct* data structure for each process:

1. num_pf is the number of page faults detected recently;
2. start_time is the system time for the first page fault in the above ‘num_pf’ page faults; and
3. privilege—the process is granted the privilege (= 1) or not (= 0).

Below are the kernel operations to determine and manage the processes exceeding the threshold page fault rates.

```
if (process p encounters page faults) {
  if (p->num_pf == 0)
    p->start_time = current system time;

  p->num_pf++;
  if (p is not in the 'high_PF_proc' list)
    if (p->num_pf > high_PF) {
      if (current system time -
          p->start_time <= 1 second)
        place p in high_PF_proc;
      p->num_pf = 0;
    }
}
```

We check the page fault rate of each process in the high_PF_proc list every second. If a process's page fault rate is lower than PF_Low, we will dynamically remove the process from the list by the following operations:

```
if (length(high_PF_proc) >= 1) {
  for each p in the list do {
    if (current system time -
      p->start_time >= 1 second) {
      if (p->num_pf/(current system time
          - p->start_time) < PF_low) {
        if (p->privilege == 1)
          p->privilege = 0;
        remove p from the list;
      }
      p->num_pf = 0;
      p->start_time = current_system_time;
    }
  }
}
```

The CPU utilization is measured every second, based on the CPU idle time. Specifically, we use (1 − idle ratio) to represent the current CPU utilization, where the idle ratio is the CPU time portion used for the idle processes in the last second. The current CPU utilization is compared with CPU_Low to determine if the system is experiencing an unacceptably low CPU utilization. The protection routine is triggered when the following three conditions are all true.

```
if ((CPU utilization < CPU_Low) &&
   (length(high_PF_proc) >= 2) &&
   (no process has been protected)) {
  for all processes in high_PF_proc
    select the least memory hungry process p;
  p->privilege = 1;
}
```

4.2. The protection routine

Privilege granting is implemented in a simple way in the kernel routine 'swap_out' presented in Section 3. The function swap_out_process(pbest, gfp_mask) will reset its 'swap_cnt' to zero if, and only if, the system fails to get an NRU page from process 'pbest', as we have shown in Section 3. A small modification in swap_out_process() will make the privilege effective; that is, we reset its 'swap_cnt' to zero even if an NRU page is obtained in the protected process. This will cause the protected process to provide at most one NRU page in each examination loop on all swappable processes. Considering that a large number of NRU pages exist in the rest of the interacting processes, such a change will effectively help the protected process build up its working set and reduce its page fault rate. Once its page fault rate is lowered satisfactorily, the protected process will be removed from the 'high_PF_proc' list and lose its privilege.

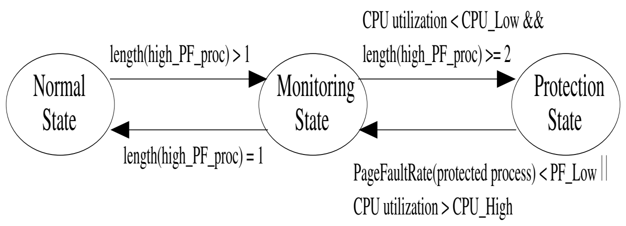

Normal State
Monitoring State
Protection State
length(high_PF_proc) > 1
length(high_PF_proc) = 1
CPU utilization < CPU_Low &&
length(high_PF_proc) >= 2
PageFaultRate(protected process) < PF_Low ||
CPU utilization > CPU_High

Figure 7. Dynamic transitions among normal, monitoring, and protection states in the improved kernel system.

4.3. State transitions in the system

The kernel memory management has the following three states with dynamic transitions.

1. *Normal state*: in this state, no monitoring activities are conducted. The system deals with page faults exactly as the original Linux kernel does. The system keeps track of the number of page faults for each process and places the process with high page fault rates in 'high_PF_proc'.
2. *Monitoring state*: in this state, the detection routine is awakened to start monitoring the CPU utilization and the page fault rates of processes in 'high_PF_proc'. If the protection condition is satisfied, the detection routine will select a qualified process for protection and go to the protection state. The system returns to the normal state when no more than one process's page fault rate is as high as the predetermined threshold.
3. *Protection state*: the protection routine will make the selected process quickly establish its working set. In the protection state, the detection routine keeps monitoring the CPU utilization and the page fault rate of each process in the list. The detection routine is deactivated and the protection state transfers to the monitoring state as soon as the protected process becomes stable and/or the CPU utilization has been sufficiently improved.

Figure 7 describes the dynamic transitions among the three states, which gives a complete description of TPF facility. When the system is normal (no page faults occur), detection and protection routines are not involved. As we have described in the implementation, the algorithm only adds limited operations for each page fault and checks several system parameters with the interval of 1 s. So, overheads involved for detection and protection one trivial compared with the CPU overhead to deal with page faults.

5. PERFORMANCE MEASUREMENTS AND ANALYSIS

5.1. Observation and measurements of TPF facility

The predetermined threshold values are set as follows: CPU_Low $=$ 40%, CPU_High $=$ 80%, PF_High $=$ 10 faults per second, PF_Low $=$ 1 fault per second. Parameter PF_High is set by the

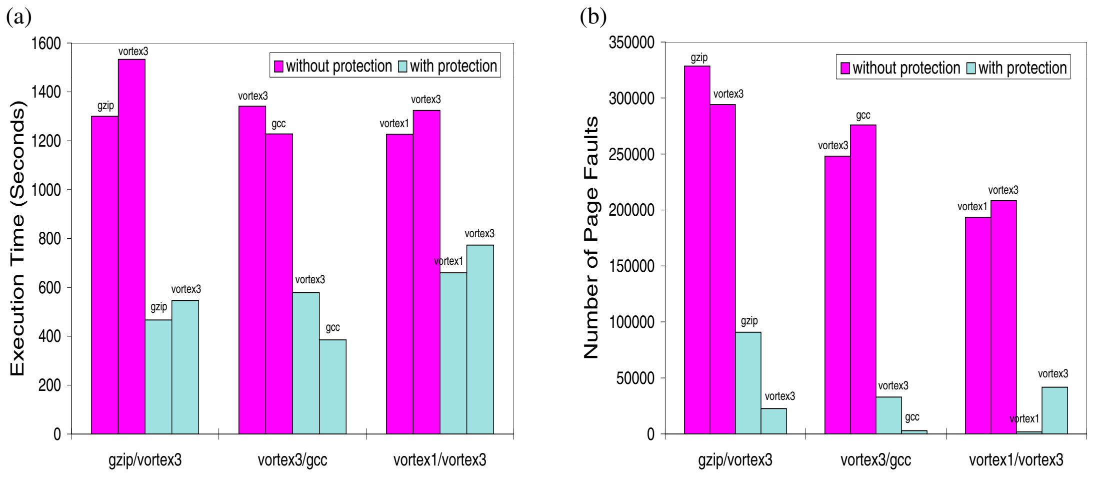


Figure 8. (a) Execution time comparisons and (b) comparisons of numbers of page faults for the three groups of program interactions in the Linux without TPF and with TPF.

following two reasons. The performance of TPF is experimentally evaluated by the three groups of the interacting programs. Each of the experiments has exactly the same setting as its counterpart conducted in Section 3, except that the TPF is implemented in the kernel.

Variable PF_High is used to identify an active process for a protection when thrashing starts. The TPF is triggered by threshold CPU_Low to select a process. The value of 10 page faults per second for a process gives two system implications. First, it tells us how much time this process spends on page swapping and other related operations. Considering the average page fault penalty time of 10–15 ms, the CPU wastes 100–150 ms or 15% of its cycles within a second for this process's page faults. Second, researchers have measured the correlations between the average page fault rate of each interacting program and the average memory oversizing percentage at that moment [6]. Based on this study, a page fault rate of 10 implies that the process demands 10% more memory space than is available.

There are several reasons for choosing a relatively small PF_High value (10 faults per second) in our system. (1) A small PF_High makes TPF responsive at an early stage in order to handle thrashing. When thrashing starts, although CPU utilization drops quickly, it takes a while for the page fault rate to reach its thrashing value (150–250 faults per second in our experiments). If we adopt a PF_High value close to the thrashing value, the protection action can be delayed. (2) Our experiments show that the value of 10 faults s^{-1} provides a proper basis for identifying a process candidate for protection. (3) Even if a non-active process, such as a process waiting for I/O, is identified and protected, TPF will quickly reduce its page fault rate. After then, another candidate will be identified.

Figures 9(a) and (b) present the memory usage measured by MAD and RSS of interacting programs gzip and vortex3, respectively, in the kernel with TPF. Figure 4 shows that thrashing between processes started as soon as gzip joined the execution at 250 s without TPF. In contrast, Figure 9 shows that TPF

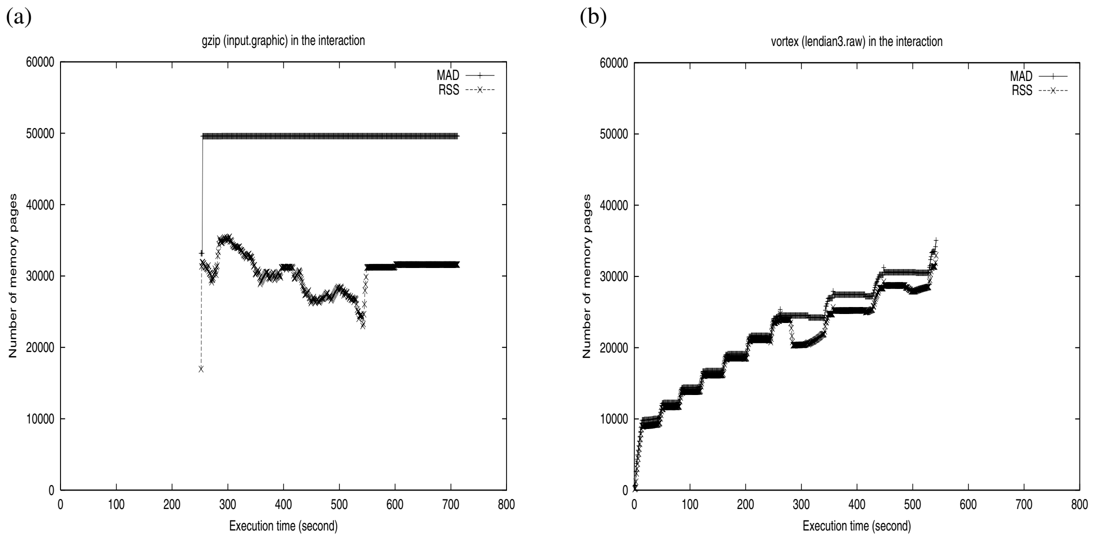


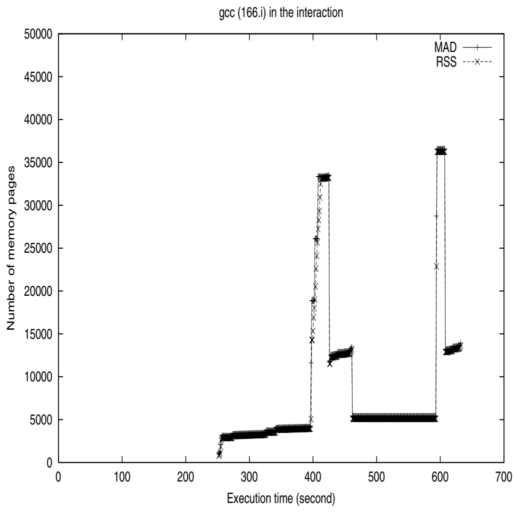
Figure 9. The memory performance of (a) gzip and (b) vortex3 during the interactions in the Kernel with TPF. Note: the execution of gzip started 250 s after that of the vortex3 program.

quickly detected the problem and went into the protection state. Because the RSS of vortex3 is close to its MAD, it was selected for protection. After the protection, its page fault rate was lowered with the establishment of its working set. Then the protection was disabled to allow the NRU pages of vortex3 to be fully utilized. This is confirmed by the small gap between MAD and RSS, which does not exist in the dedicated execution (see Figure 3). In the experiment we observed that TPF had to come back and forth during the program interaction over 10 times to help vortex3 establish working sets. This is because the program vortex is not strong enough to keep its established working set with the competition of gzip. Even for this type of program, TPF demonstrates its effectiveness. The numbers of page faults and execution time of vortex3 are reduced by 72 and 92%, respectively (see Figure 8).

The performance improvement for gzip is also significant. Its number of page faults and execution time are reduced by 72 and 64%, respectively. Intuitively, its performance should have been degraded because it contributed more memory space to vortex3 for building up its working set enforced by TPF. But this is not the case for two reasons. First, under the protection of TPF, vortex3 had an early completion. Then gzip could run without memory competition and use CPU cycles solely. Second, under the protection of TPF, vortex3 could greatly reduce its page fault rate, which made gzip utilize most of the I/O bandwidth and reduced page fault penalty.

Figures 10(a) and (b) present the memory usage measured by MAD and RSS of interacting programs gcc and vortex3, respectively, in the kernel with TPF. At 397 s, memory demand from gcc rapidly rose, and both programs started page faults due to memory shortage. The thrashing significantly lowered CPU utilization, which triggered TPF to take action. Because gcc demanded memory gradually, and

(a)



(b)

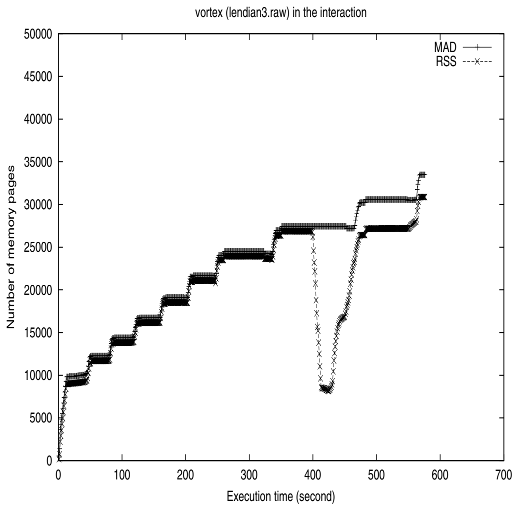
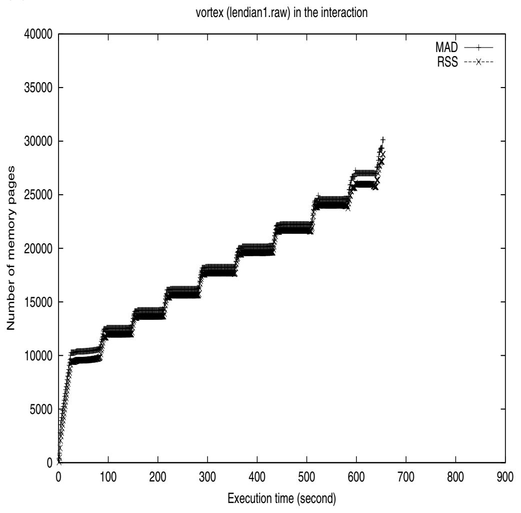


Figure 10. The memory performance of (a) gcc and (b) vortex3 during the interactions in the kernel with TPF. Note: the execution of the gcc program started 250 s after that of the votex3 program.

(a)



(b)

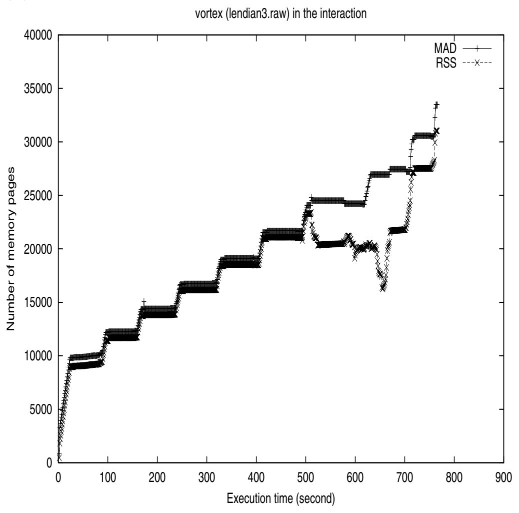


Figure 11. The memory performance of (a) vortex1 and (b) vortex3 in the kernel with TPF.

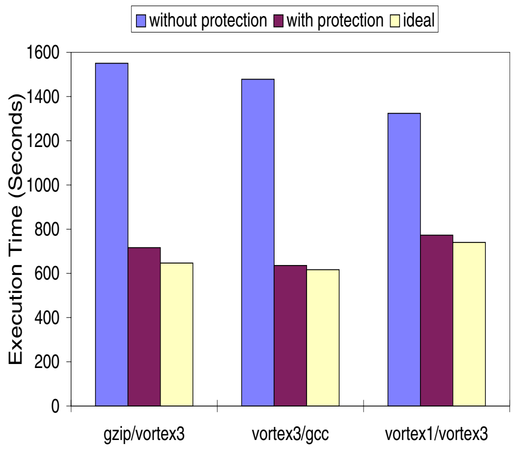


Figure 12. Comparison of total interaction execution times for the three groups of program interactions in the Linux with TPF, without TPF and the ideal interaction times.

kept the gap between MAD and RSS small, gcc was selected for protection on its rising slope of the first MAD spike by TPF. The memory is dynamically allocated between two processes to ensure a reasonable level of CPU utilization. The period the system remains in the protection state is very limited, thus memory utilization is maintained. TPF successfully smoothed out the peak in memory load that might otherwise have caused the system to thrash. Compared with the same run in the original Linux kernel, the execution times of programs vortex3 and gcc are reduced by 57 and 69%, respectively; the numbers of page faults of programs gcc and vortex3 are reduced by 87 and 99%, respectively (see Figure 8).

Figures 11(a) and (b) present the memory usage measured by MAD and RSS of interacting programs vortex1 and vortex3, respectively, in the kernel with TPF. During the interactions at the execution time of 433 s, both programs started page faults due to memory shortage. The program vortex1 was then protected by TPF. We observed that vortex1 easily held its working set thereafter and only a small amount of time of TPF involvement was needed. This is because vortex1 and vortex3 have similar memory access rates and patterns. Thus once vortex1 was given privilege to establish its working set, it would keep the working set by frequently using it. In contrast to the performance seen in Figure 6, a small correction from TPF could make a big difference in multiprogramming. Compared with the same execution in the original Linux kernel, the execution times of programs vortex1 and vortex3 are reduced by 46 and 42%, respectively; the numbers of page faults are reduced by 99 and 80%, respectively (see Figure 8).

Figure 12 compares the total execution times for the three groups of interacting programs in Linux with and without TPF. We assume the execution times of each pair of programs under the same multiprogramming condition with sufficient memory space as the ideal interaction execution time. Figure 12 shows that the total interacting execution times in the kernel with TPF for the three groups are significantly smaller than those in the kernel without TPF, and are very close to the ideal execution times. These experiments also indicate that TPF has little runtime overhead.

5.2. Experiences with TPF in the multiprogramming environment

1. *Under what conditions, does thrashing happen in a multiprogramming environment?* Our experiments show that VMs in Linux can normally keep a reasonable CPU utilization even under heavy workload, adapting the variance of memory demands, access patterns and access rates of different processes. A process that can frequently access its working set in execution interactions has a strong position for memory space competitions during interactions. However, under the following three conditions, thrashing can be triggered.

    - The memory demands of one interacting program have sudden 'spikes'. Case studies in Figures 4 and 5 show such examples.
    - The variance of memory demands, access patterns, and access rates of interacting processes are similar. Case studies in Figure 6 show such examples.
    - Serious memory shortage happens in the system.

2. *For what cases is TPF most effective?* TPF is most effective in the first two cases discussed above. In other words, TPF is able to quickly resolve thrashing for interacting programs having dynamically changing memory demands. We have shown that TPF is highly responsive to increasing CPU utilization and to stopping thrashing by adapting page replacement to memory allocations. In addition, the scheduling action from TPF has little intervention in the system and the multiprogramming environment because the protection period is very short, but it is effective in leading the system back to normal.
3. *For what cases is TPF ineffective?* If the memory shortage problem is too serious in a multiprogramming environment, the selected process may build up and hold its working set in memory at the cost of obtaining most of the memory space of other processes. Although TPF can still cause CPU cycles to be effectively utilized, the CPU overhead serving page faults of other processes will significantly increase, and I/O channels may become heavily loaded due to a large amount of page faults. As a result, the protected process will not run smoothly. Under such conditions, the load control has to be used to swap a process for releasing memory space. With the support of TPF, the load control facility will be used only when it is truly necessary.
4. *How do the threshold parameters affect the performance of TPF?* We summarize the relationships between the four threshold parameters (see Section 3.1) and the effectiveness of TPF. A large value of CPU_Low and a small value of PF_High will make TPF more responsive to system thrashing. In contrast, a large value of CPU_High and a small value of PF_Low will make the identified process stay longer in the protection state after it enters the state. Thus, adjusting these parameters is equivalent to changing the extent of TPF intervention to the system. The parameters are set based only on system requirements; they are not dependent on application program natures. For example, for systems with high I/O bandwidths (e.g. parallel disk arrays), larger values can be set for PF_High and PF_Low, because page faults can be resolved quickly. In our experiments we found that the performance of TPF was quite stable within a large range of parameter values.

6. RELATED WORK

Improvement of CPU and memory utilizations has been a fundamental consideration in the design of operating systems. Extensive research on thrashing had been conducted since the 1960s. Among the proposed policies the most influential one is the working set policy, which was able to thoroughly protect against thrashing while keeping high CPU utilization. The working set policy also provides a solution at the page replacement level.

6.1. The working set model and its implementation issues

Denning proposes a working set model [7–9] to estimate the current memory demand of a running program in the system. A working set of a program is a set of its recently used pages. Specifically, at virtual time t, the program's working set $W_t(\theta)$, is the subset of all pages of the program, which has been referenced in the previous θ virtual time units (working set window). The task's virtual time is a measure of the duration the program has control of the processor and is executing instructions without interruptions. In practice, NAP(t) is equivalent to $W_t(\theta)$ with a 1 s sampling interval of dedicated execution, where NAP is the number of accessed pages (defined in Section 3.1) at a given time t. The working set model ensures that the same program with the same input data would have the same locality measurements, which is independent of the memory size, the multiprogramming level, and the scheduler policy used. A working set policy is used to ensure no pages in the working set of a running program will be replaced. Assuming that priorities among the processes exist, once there is a request for free pages, but they are not available, the processes with the lowest priority has to produce a victim page for replacement. This implies that an active process with the lowest priority may not fully allocate its working set. Since the I/O time caused by page faults is excluded in the working set model, the working set replacement algorithm can theoretically eliminate the thrashing caused by chaotic memory competition. Comparatively, other global policies like LRU approximations (two-handed clock, FIFO with second chance) used in the currently popular Unix-like operating systems, are highly susceptible to thrashing, because a program's resident set depends on many factors besides its own locality. Our experimental observations are consistent with the conclusions in the cited work on working set models.

A major difficulty in implementing the working set model in a modern computer system is its implementation overhead scaling with the capacity of CPU and memory. The working set model can be implemented by either hardware or software. To build a hardware for working set detection, registers (one per page), timers, and circuits are needed to add to the memory. Since these additions would complicate chip design, no major chip vendor has been willing to implement the working set model.

Compared with the approach of only associating a page-reference bit with each page frame to support LRU related page replacement policies in Linux and Unix systems, an implementation of the working set detector is more expensive. With the increase of CPU speed and memory capacity, and with an increasing amount of memory-intensive workloads in applications, the number of active pages owned by a process has dramatically increased, which has become a major reason to limit such an implementation. In addition, since system thrashing is considered an exceptional event, it may be difficult to convince computer vendors to provide hardware support for the working set model. Instead, the computer architects prefer to adopt some brute-force methods as exceptional handlers, such as to release memory space by urgently removing some processes.

Implementing the working set detector by system software, we need to routinely update a software counter associated with each page frame. Since monitoring a huge amount of page frames is part of routine operations in memory management, it would affect the system performance even when the system functions normally.

Although a number of good approximations exist, the implementation cost of the working set policy may limit its direct usage in modern computer systems. However, this model has given us two important motivations in memory system designs and implementation. First, being a local memory policy, the working set policy has inherent load controls, and needs no special, additional mechanism to deal with thrashing. This certainly saves the cost of an additional mechanism to stabilize global LRU policy. In principle, the working set model can be turned by adjusting its window size, so that it would operate the system at a nearly optimal MPL. In other words, the highest possible system throughput would be reached. Second, the working set policy employs 'feedforward control', rather than 'feedback control', which means that a working set does not have to react to thrashing, but avoids thrashing in advance. Motivated by our comparisons and investigations of page replacement implementations in existing system kernels, and guided by the principle of the working set model, we propose the TPF, which is not part of the routine operations in memory management, but is only triggered at an early stage of thrashing to effectively stop the thrashing or significantly delay the load controls. In addition, when the system is not in danger of thrashing, unlike the working set model, TPF is inactive, avoiding the working set sampling overhead.

6.2. Other related work

Studies of page replacement policies have a direct impact on memory utilization, which have continued for several decades (e.g. a representative and early work in [10], and recent work in [11,12]). The goal of an optimal page replacement is to achieve efficient memory usage by only replacing those pages not used in the near future when available memory is not sufficient, reducing the number of page faults. In a single-programming environment, these proposed methods address both concerns of CPU and memory utilization since any extra page faults due to low memory utilization will make the CPU stall. However, the system thrashing issue in a multiprogramming environment can not be fully addressed by the cited work due to the conflicting interests between CPU and memory utilization.

In the multiprogramming context, existing systems mainly apply two methods to eliminate thrashing. One is local replacement, another is load control. A local replacement policy requires that the paging system select pages for a program only from its allocated memory space when no free pages can be found in their memory allotments. Unlike a global replacement policy, a local policy needs a memory allocation scheme to satisfy the needs of each program. Two commonly used policies are equal and proportional allocations, which cannot capture the dynamical changing memory demand of each program [13]. As a result, memory space may not be well utilized. In contrast, an allocation policy dynamically adapting to the demands of individual programs will shift the scheme to global replacement. VMS [14] is a representative operating system using a local replacement policy. Memory is partitioned into multiple independent areas, each of which is localized to a collection of processes that compete with one another for memory. Unfortunately, this scheme can be difficult to administer [15]. Researchers and system practitioners seem to have agreed that a local policy is not an effective solution for virtual memory management. Our TPF is built on a global replacement policy.

The objective of load control is to lower the MPL by physically reducing the number of interacting processes. A commonly used load control mechanism is to suspend/reactivate processes, even swapping out/in processes to free more memory space, when thrashing is detected. The 4.4 BSD operating system [2], the AIX system in the IBM RS/6000 [16], and HP-UX 10.0 in HP 9000 [17] are examples that adopt this method. In addition, the HP-UX system provides a 'serialize()' command to run the processes one at a time when thrashing is detected. In contrast, TPF protects the system from thrashing at the page replacement level. Memory allocation scheduling at this level allows us to carefully consider the tradeoff between CPU and memory utilization.

7. CONCLUSION

We have investigated the risk of system thrashing in page replacement implementation by examining the Linux kernel code of versions 2.0, 2.2, and 2.4, and running interacting benchmark programs in a Linux system. Our study indicates that this risk is rooted in conflicting interests of requirements on CPU and memory utilizations in a multiprogramming environment. We have experimentally observed several system thrashing cases when processes dynamically and competitively demand memory allocations, which causes low CPU utilization and long execution time delays, and eventually threatens system stability.

We have proposed TPF and implemented it in the Linux kernel to prevent the system from thrashing among interacting processes, and to improve the CPU utilization under heavy load. TPF will be awakened when the CPU utilization is lower than a predetermined threshold, and when the page fault rates of more than one interacting process exceed a threshold. TPF then grants privilege to an identified process to limit its contributions of NRU pages. We create a simple kernel monitoring routine in TPF to dynamically identify an interacting process that highly deserves temporary protection. The routine also monitors whether the identified process has satisfactorily lowered its page fault rate after the protection. If so, its privilege will be disabled to let it equally participate in contributing NRU pages with other processes.

Conducting experiments and performance evaluation, we show that the TPF facility can effectively provide thrashing protection without negative effects to overall system performance for three reasons: (1) the privilege is granted only when a thrashing problem is detected; (2) although the protected process could lower the memory usage of the rest of the interacting processes for a short period of time, the system will soon become stable by the protection; and (3) TPF is simple to implement with little overhead in the Linux kernel. Because the conflicting interests between CPU and memory utilization are inherent in global page replacement, and our solution is targeted at regulating the conflicts through tuning page replacement, we believe that the TPF idea is applicable to VMs of other UNIX-like systems.

ACKNOWLEDGEMENTS

The work is a part of an independent research project sponsored by the National Science Foundation for program directors and visiting scientists. We thank Phil Kearns for providing a kernel programming environment. We thank Bill Bynum for reading the paper and for his suggestions. We thank the anonymous referees for their insightful and constructive comments which have improved the quality and readability of this paper. Finally, we are grateful to Professor Peter J. Denning for his expert views and advice on this work.

REFERENCES

1. Peterson JL, Silberschatz A. *Operating System Concepts* (2nd edn). Addison-Wesley: Reading, MA, 1985.
2. McKusick MK, Bostic K, Karels MJ, Quarterman JS. Memory management. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley: Reading, MA, 1996; 117–190.
3. Tanenbaum AS, Woodhull AS. Memory management. *Operating Systems, Design and Implementation*. Prentice-Hall: Englewood Cliffs, NJ, 1997; 309–400.
4. Maxwell S. *Linux Core Kernel Commentary*. Coriolis Open Press: Scottsdale, AZ, 1999.
5. van Riel R. Page replacement in Linux 2.4 memory management. *Proceedings of USENIX Annual Technical Conference* (FREENIX track), Boston, MA, June 2001.
6. Xiao L, Chen S, Zhang X. Dynamic cluster resource allocations for jobs with known and unknown memory demands. *IEEE Transactions on Parallel and Distributed Systems* 2002; **13**(3).
7. Denning PJ. The working set model for program behavior. *Communications of the ACM* 1968; **11**(5):323–333.
8. Denning PJ. Working sets past and present. *IEEE Transactions on Software Engineering* 1980; **6**:64–84.
9. Denning PJ. Before memory was virtual. *In the Beginning: Personal Recollections of Software Pioneers*, Glass R (ed.). IEEE Press: Los Alamitos, CA, 1997 (also available at http://www.cne.gmu.edu/pjd/PUBS/bvm.pdf).
10. Aho AV, Denning PJ, Ullman JD. Principles of optimal page replacement. *Journal of ACM* 1971; **18**(1):80–93.
11. Glass G, Cao P. Adaptive page replacement based on memory reference behavior. *Proceedings of 1997 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1997; 115–126.
12. Smaragdakis Y, Kaplan S, Wilson P. EELRU: Simple and effective adaptive page replacement. *Proceedings of 1999 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1999; 122–133.
13. Coffman EG Jr., Ryan TA. A study of storage partitioning using a mathematical model of locality. *Communications of the ACM* 1972; **15**(3):185–190.
14. Kenah LJ, Bate SF. *VAX/VMS Internals and Data Structures*. Digital Press: Bedford, MA, 1984.
15. Lazowska ED, Kelsey JM. Notes on tuning VAX/VMS. *Technical Report 78-12-01*, Department of Computer Science, University of Washington, December 1978.
16. IBM Corporation. *AIX Versions 3.2 and 4 Performance Tuning Guide*, April 1996.
17. HP Corporation. *HP-UX 10.0 Memory Management White Paper*, January 1995.