Available online at www.sciencedirect.com

SCIENCE DIRECT®

Performance Evaluation 60 (2005) 5–29

PERFORMANCE EVALUATION
An International Journal

www.elsevier.com/locate/peva

Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems

Song Jiang[a, 1], Xiaodong Zhang[b, *]

[a] *Los Alamos National Laboratory, Los Alamos, NM 87545, USA*
[b] *Department of Computer Science, College of William and Mary, Williamsburg, VA 23187, USA*

Abstract

Most computer systems use a global page replacement policy based on the LRU principle to approximately select a Least Recently Used page for a replacement in the entire user memory space. During execution interactions, a memory page can be marked as LRU even when its program is conducting page faults. We define the LRU pages under such a condition as *false LRU* pages because these LRU pages are not produced by program memory reference delays, which is inconsistent with the LRU principle. False LRU pages can significantly increase page faults, even cause system thrashing. This poses a more serious risk in a large parallel systems with distributed memories because of the existence of coordination among processes running on individual node. In the case, the process thrashing in a single node or a small number of nodes could severely affect other nodes running coordinating processes, even crash the whole system. In this paper, we focus on how to improve the page replacement algorithm running on one node.

After a careful study on characterizing the memory usage and the thrashing behaviors in the multi-programming system using LRU replacement. we propose an LRU replacement alternative, called *token-ordered LRU*, to eliminate or reduce the unnecessary page faults by effectively ordering and scheduling memory space allocations. Compared with traditional thrashing protection mechanisms such as load control, our policy allows more processes to keep running to support synchronous distributed process computing. We have implemented the token-ordered LRU algorithm in a Linux kernel to show its effectiveness.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Process thrashing; Global LRU replacement; Load control; Performance evaluation

* Corresponding author. Tel.: +1 757 221 3458; fax: +1 757 221 1717.
[1] Tel.: +1 505 606 0308; fax: +1 505 667 1126.
E-mail addresses: sjiang@lanl.gov (S. Jiang); zhang@cs.wm.edu (X. Zhang).

0166-5316/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.peva.2004.10.002

1. Introduction

A major issue in resource management for high-end computing systems, such as large parallel systems of distributed memory is on how to well utilize the local memory on each computing node to maintain sustained high performance of the entire system. This issue is also a serious concern for tightly coupled super server, and even for a powerful desktop computing server running memory intensive applications. Process thrashing is the most disastrous issue in the local memory management. Because of the existence of coordination among processes at different nodes, process thrashing in a single node or a small number of nodes could severely affect other nodes running coordinating processes, even crash the whole system. For large-scale parallel systems with a large number of nodes, the probability and significance of the problem could become much more serious.

Because there could be unpredictable memory demands from multiple processes running in the same local memory space, and limited memory size, process thrashing could unpredictably happen at any node in an indeterministic way. In a parallel system without inter-node memory sharing, replaced pages have to temporarily write into or read from hard disks, leaving the affected nodes being able to complete little useful computing work. Thus a local memory management with strong resistance to process thrashing is essential to the performance of whole parallel system. In this paper, we focus on how to improve the page replacement algorithm running on one node and to prevent the local system from thrashing in an efficient way to support the whole parallel system.

Virtual memory systems manage space sharing among interacting programs[2] by page replacements when the demanded memory allocations are larger than the available memory space. A commonly used replacement policy in virtual memory management is the global Least Recent Used (LRU) replacement, which selects an LRU memory page for replacement throughout the entire user memory space of the system. Many of computing practitioners may have experienced the following program execution difficulties in multiprocess systems. When the accumulated memory demands of multiple interacting programs exceed the available user memory space to a certain degree, the system starts thrashing—none of the programs are able to establish their working sets, causing a large number of page faults in the system, low CPU utilizations, and a long delay for each program. Although a large number of cycles are wasted during program interactions, people seem to have accepted this reality, and to believe these additional cycles are unavoidable due to the memory resource shortage and due to the fairness requirement among interacting programs.

Intuitively, all interacting programs are treated equally by the LRU policy, because no single program has priority over others. Looking into the way an LRU replacement policy is implemented will give us more insights on its effectiveness. An allocated memory page of a program will become a replacement candidate if the page has not been accessed for a certain period of time under two conditions: (1) the program does not need to access the page; and (2) the program is conducting page faults (a sleeping process) so that it is not able to access the page although it might have done so without the page faults. We call the LRU pages generated on the first condition *true LRU pages*, and those on the second condition *false LRU pages*. These false LRU pages are produced by the time delay of page faults, not by the access delay of the program. The LRU principle is not maintained. However, LRU page re-

[2] By interacting programs, we refer to the processes that share the same memory space and interact with each other through a global replacement algorithm.

placement implementations do not discriminate between these two types of LRU pages, and treat them equally!

Whenever page faults occur due to memory shortage in a multiprogramming environment, false LRU pages of a program can be generated, which will weaken the ability of the program to achieve its working set. For example, if a program does not access the already obtained memory pages on the false LRU condition, these pages may become replacement candidates (LRU pages) when the memory space is being demanded by other interacting programs. When the program is ready to use these pages in its execution turn, these LRU pages may have been replaced to satisfy requested allocations of other programs. The program then has to ask the virtual memory system to retrieve these pages by replacing LRU pages of others, possibly generating false LRU pages for other programs. The false LRU pages may be cascaded among the interacting programs, eventually causing system thrashing.

In order to address the problems caused by the false LRU condition in global page replacement, we propose a token-ordered LRU policy. The basic idea is to set a token in the system. The token is taken by one of the programs when page faults occur. The system eliminates the false LRU pages from the program holding the token to allow it to quickly establish its working set. By giving this privilege to a program during the interactions, we are able to reduce the total number of false LRU pages and to transform the chaotic order of page usages to an arranged order.

2. Backgrounds of thrashing protections

Researchers in the operating system field have proposed several schemes to protect system from thrashing during program interactions, and some of them are implemented in the practical systems. The framework of *local page replacements* [2] and *working set models* [7], have been proposed to address the issue. After thrashing is detected, *load controls* [8] can be used to eliminate it. In this section, we will briefly overview these schemes and techniques, and discuss their limitations. These related studies have motivated us to propose and implement the token-ordered scheme to address the limitations.

2.1. Local page replacement

Although our targeted paging system uses global page replacement, local page replacement has been a proposed solution to protect thrashing for program interactions. A local replacement requires that the paging system select victim pages for a program only from its allocated memory space when no free pages can be found in its memory allotment. Unlike the global replacement policy, the local policy needs a memory allocation scheme to satisfy the need of each program. Two commonly used policies are equal and proportional allocations, which can not capture dynamically changing memory demand of each program [5]. As a result, the memory space may not be well utilized. On the other hand, an allocation dynamically adapting to the demand of individual program will shift the scheme to the global replacement. VMS [20] is a representative operating system using a local replacement policy. The memory is partitioned into multiple independent areas, each of which is localized to a collection of processes that compete with one another for memory space. Unfortunately, this scheme can be difficult to administer [18]. Researchers and system practitioners seem to have agreed that a local policy is not an effective solution for virtual memory management.

2.2. Working set models

Denning [7] proposes a working set model to measure 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. A working set replacement algorithm is used to ensure no pages in the working set of a running program will be replaced [9]. Since the I/O time caused by page faults is excluded in the working set model, the working set replacement algorithm can theoretically eliminate thrashing caused by chaotic memory competition. However, the implementation of this model is extremely expensive because working set monitoring is required for each individual program based on its virtual time [19]. The affordable LRU approximations of the working set algorithm, such as clock, FIFO with second chance have to give up considering the "virtual time" for each individual program when determining the working set. This approximation leaves a loophole to introduce false LRU pages. One contribution of our token-ordered LRU replacement scheme is to approximate the "true" working set for the identified program to eliminate its false LRU pages.

2.3. Load controls

A commonly used method to protect systems from thrashing is load control, which adjusts the memory demands from multiple processes by changing the multiprogramming level (MPL), or the number of active processes in the system. It suspends/reactivates, even swaps out/in processes to control memory demands after thrashing is detected. The 4.4 BSD operating system [22], AIX system in the IBM RS/6000 [15], HP-UX 10.0 in HP 9000 [14] are the examples to adopt this method. In addition, HP-UX system provides a "serialize()" command to run the processes once at a time after thrashing is detected.

2.4. Why is a lightweight thrashing prevention mechanism desired?

The most destructive aspect of thrashing is that, although thrashing may have been triggered by a brief, random peak in workloads (e.g. all of the users of a system happen to press the Enter keys at the same second), the system might continue thrashing for an indefinitely long time. Because thrashing is often a result of a sudden spike in workloads, a lightweight, dynamic protection mechanism is more desirable than a brute-force action, such as program suspension or even a program removal in a parallel computing environment. This is because suspension-based load controls have several limitations. First, a suspension/reactivation scheme simply stops some processes from functioning, which could cause synchronous processes in other nodes severely delayed. This will help spread the illness of the thrashing node to other related nodes. Second, a suspension/reactivation scheme is detection-based. Before certain conditions are detected and the suspension/reactivation actions are taken, the system is thrashing or its memory is under-utilized in a time period. Third, in a dynamic program interaction environment, a short moment of low/high free memory or page fault rates may not mean thrashing is immediately coming/leaving. Thus, it is hard to determine when a program suspension/reactivation is initiated with the dynamically changing memory requirements from active programs. A wrong decision will degrade system performance. Finally, when a program is suspended, a large portion of its entire working set can be replaced for other running programs. Re-establishing the working set after reactivating its execution, particularly for a large suspended program, could involve a significant amount of additional overheads.

It is noted that we do not treat thrashing as a long-term pathological system condition of systems with limited memory in the paper. We believe that the final solution to constant thrashings in a system due to memory shortage is a system upgrade, and a *real* thrashing due to a serious memory shortage can only be removed through swapping out processes to reduce memory demands. Using the token-ordered replacement in the first place, we are able to eliminate thrashing in its early stage, significantly delaying the usage of load controls. As a proactive scheme, our token-ordered LRU tries to attain the same goal as load controls in thrashing protection without the specific limitations of load controls. With the token-ordered replacement and load controls guarding at two different levels and two different stages, system performance will become more stable and cost-effective.

3. Experimental environment

3.1. Workloads

We have selected 10 memory-intensive application programs, five of which are from SPEC 2000 (*apsi*, *gcc*, *gzip*, *mcf*, and *vortex*), and the other five are from data reordering, matrix computation, and graphics applications, which are briefly described as follows. All of these programs are both CPU-intensive and memory-intensive:

- *Bit-reversals* (bit-r): This program conducts data reordering operations, which are required in many Fast Fourier Transform (FFT) algorithms [27].
- *Matrix multiplication* (m-m): This is a standard matrix multiplication program [24].
- *Merge-sort* (m-sort): This is a standard merge sorting program [26].
- *LU decomposition* (LU): This is a standard matrix LU decomposition program for solving linear systems [24].
- *Cell-projection volume rendering for the flow of an aircraft wing* (r-wing): The input data of the volume rendering program is the flow over an aircraft wing with an attached missile, with 500,000 cells. Ma and Crockett [21] have developed a parallel cell-projection algorithm. We used its sequential version in our experiments.

3.2. Experimental system support

Our performance evaluation is based on experimental measurements. The machine we have used for all experiments is a Pentium II at 400 MHz with a physical memory space of 384 MB. The operating system is Red Hat Linux release 6.1 with the kernel 2.2.14. Program memory space is allocated in an unit of 4 KB page. The disk is an IBM Hercules with its capacity of 8450 MB.

When memory related activities in a program execution occur, such as memory accesses and page faults, the system kernel is heavily involved. To gain insights on the memory system behaviors of application programs, we have monitored program executions at the kernel level, and carefully made some simple instrumentation in the system. Our monitor program has two functions: user memory space adjustment and system data collection. In order to flexibly adjust the available memory space for user programs in the experiments, the monitor program can serve as a memory-adjustment process requesting a memory space of a fixed size, which is excluded from the page replacement. The available user memory space can be flexibly adjusted by running the memory-adjustment process with different fixed sizes of memory demands. The difference between the physical user memory space and the demanded memory size of the memory-adjustment process is the available user space in our experiments. The memory-adjustment process will not affect our experiment measurements. This is because (1) it consumes few CPU cycles; and (2) its resident memory is excluded from the global page replacement scope. So its memory usage has no interactions with application programs.

The monitoring program dynamically collects following memory system status quanta at every other time interval of 1 s during the execution of programs:

- *Memory allocation demand* (MAD): is the total amount of requested memory space reflected in the page table of a program 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 effects on the program executions.
- *Resident set size* (RSS): is the total amount of physical memory used by a program in pages, and can be obtained from *task_struct*.
- *Number of page faults* (NPF): is the number of page faults of a program, and can also be obtained from *task_struct*. There are two types of page faults for each program: 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 the physical memory. 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 the memory and has to be fetched from a disk. We only collect major page fault events for each program.
- *Number of accessed pages* (NAP): is the number of accessed pages by a program within the time interval of 1 s. This is collected by a simple system instrumentation. During a program execution, a system routine is periodically invoked to examine all the reference bits in the page table of a specific program.

Using the system facilities described above, we first run each program in a dedicated environment to observe the memory access behavior without major page faults and page replacements (the demanded memory space is smaller than the available user space). Table 1 presents the basic experimental results of the 10 programs, where the “description” gives the application nature of each program, the “input file/size” is the input file names from SPEC2000 benchmarks, or the number of entries of the input data of the other five application programs, the “max MAD” gives the maximum size of the MADs during its execution, the “lifetime” is the execution time of each program. These measurements represent the mean of five runs. The variation coefficients calculated by the ratio of the standard deviation to the mean are less than 0.01.

3.3. Memory access behavior in dedicated environment

The memory usage patterns of all 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 mem-

Table 1
Execution performance and memory related data of the 10 benchmark programs, where the program names with * are SPEC 2000 benchmarks

Programs	Description	Input file/size	Max MAD (MB)	Lifetime (s)
*apsi	Climate modeling	apsi.in	196.0	2628.3
*gcc	Optimized C compiler	166.i	145.0	218.7
*gzip	Data compression	input.graphic	197.4	248.7
*mcf	Combinatorial optimization	inp.in	79.2	975.9
*vortex	Database	lendian1.raw	115.0	342.3
*vortex	Database	lendian3.raw	131.2	398.0
bit-r	Data reordering	2^{25}	131.3	326.1
m-m	Matrix multiplication	1800^2	76.2	1430.3
m-sort	Merge sort	2^{24}	131.4	58.3
LU	LU decomposition	2000^2	161.2	98.0
r-wing	Volume rendering	500,000	48.9	60.8

ory usage curves: the memory allocation demand (MAD), the resident set size (RSS), and the number of accessed pages (NAP). The memory usage curves of the 10 benchmark programs measured by MAD, RSS, and NAP are presented in Fig. 1 (*apsi* and *gcc*), Fig. 2 (*gzip* and *mcf*), Fig. 3 (*vortex1*[3] and *bit-r*), Fig. 4 (*m-m* and *m-sort*), and Fig. 5 (*LU* and *r-wing*). With regard to the development of memory demands, the memory usage patterns exhibited in the 10 graphs are classified in three types:

1. *Quickly acquiring memory allotments*: This type of programs demands stable memory allocations from the beginning of program executions. When the available space is sufficient, they can quickly acquire their allotments in the early stage of their executions. Programs *apsi*, *bit-r*, *gzip*, and *m-m* belong to this type. Among them, *apsi*, *m-m* have more regular accesses to their allotments, and also have stable working set sizes hinted by their NAP curves.
2. *Gradually acquiring memory allotments*: This type of programs gradually increases the memory allotments as their executions progress, and access the data sets regularly in each stage until their executions complete. When the available space is sufficient, their RSS sizes in each time interval form stair climbing curves as their executions proceed. Programs *mcf*, *m-sort*, *r-wing*, and *vortex* belong to this type. However, their access frequencies on the allotments could be different. For example, the NAP curves indicate that *vortex* accesses its RSS memory more vigorously than *mcf* does, implying that its RSS is closer to its realistic working set size than the RSS of *mcf*.
3. *Non-regularly changing memory allotments*: This type of programs has non-regular memory demands in their life times of executions. Their demands on memory sizes are changed dynamically with high variations. When the available space is sufficient, there are multiple ups and downs in their RSS curves as their executions proceed. Programs *gcc* and *LU* belong to this type.

[3] *vortex1* is the vortex with input file “lendian1.raw”, *vortex3* is the vortex with input file “lendian3.raw”.

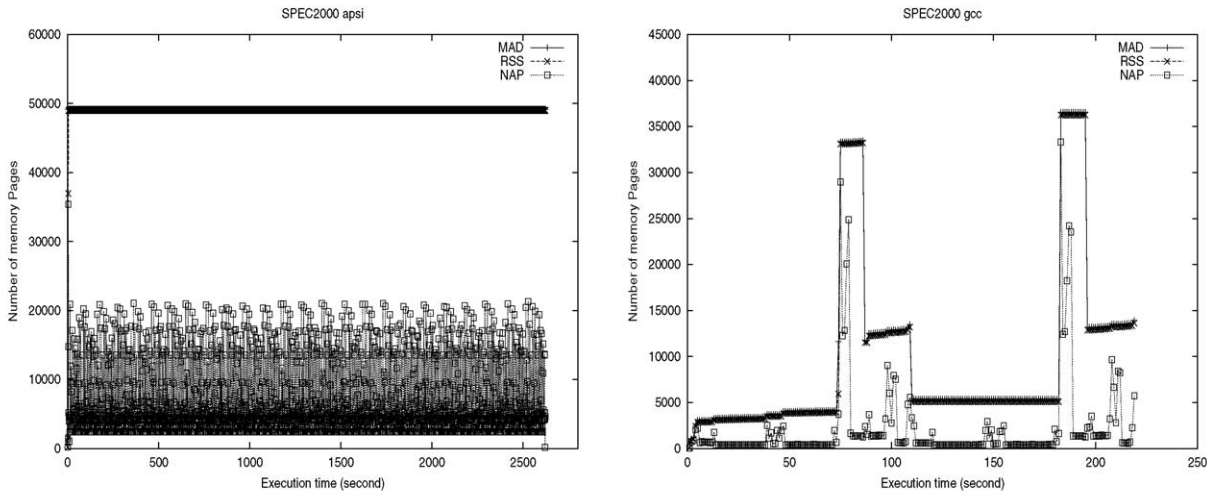


Fig. 1. The memory performance of apsi (left) and gcc (right) in a dedicated environment.

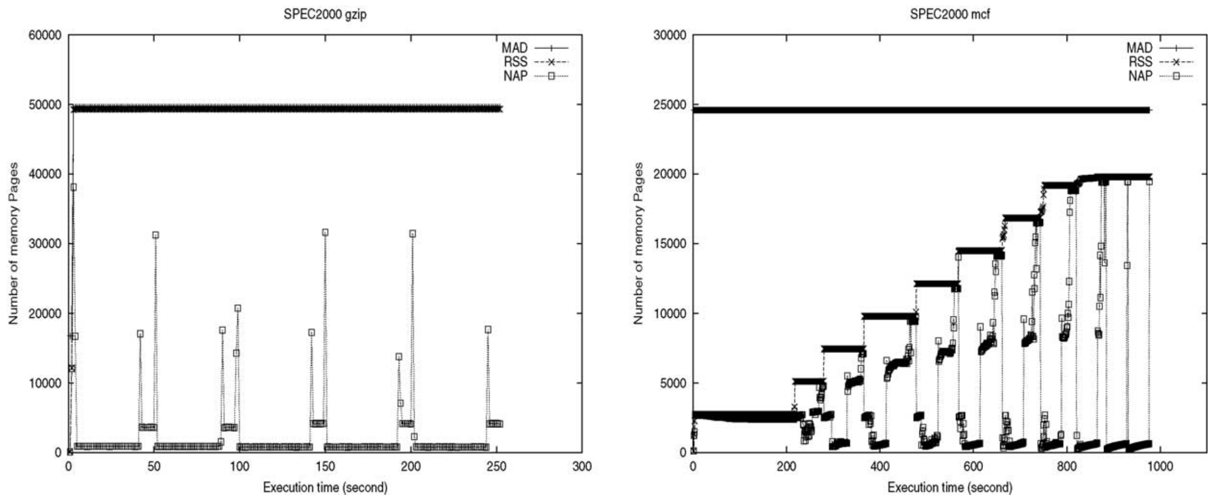
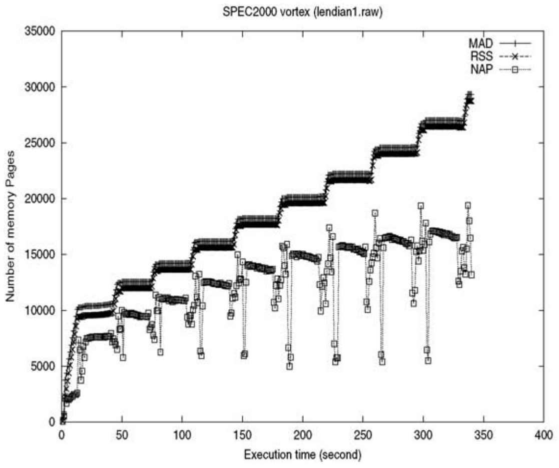


Fig. 2. The memory performance of gzip (left) and mcf (right) in a dedicated environment.

4. Memory performance of different types of program interactions

4.1. Performance metrics

We use *slowdown* to measure the performance degradation of an interacting program, which is defined as the ratio between the execution time of an interacting program and its execution time in a dedicated environment without major page faults. Major contributions to slowdown come from the penalties of page faults, CPU sharing, context switch and monitoring activity overheads. We found that context switch and monitoring activity overheads are trivial in our measurements.



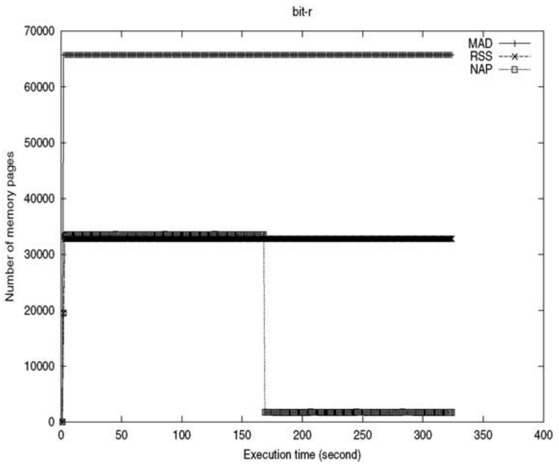
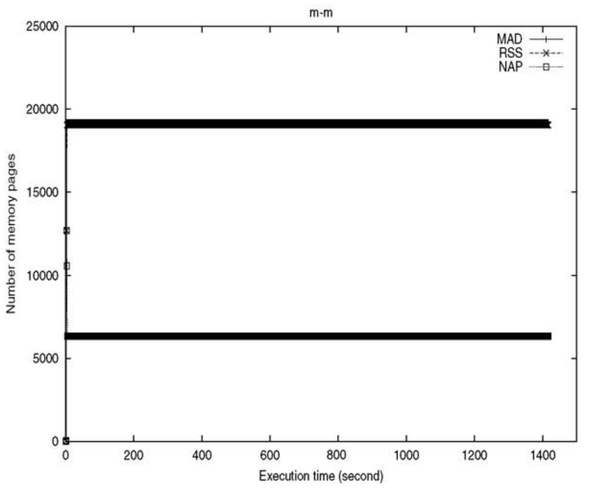


Fig. 3. The memory performance of vortex1 (left) and bit-r (right) in a dedicated environment.

4.2. Memory performance of program interactions

Recall that we have classified three types of memory demand patterns in programs, namely, type 1: quickly acquiring memory allotments; type 2: gradually acquiring memory allotments; and type 3: non-regularly changing memory allotments. There are seven typical group combinations for program interactions from the these three types: type 1 and type 2 (group 1), type 1 and type 3 (group 2), type 2 and type 3 (group 3), three types together (group 4), multiple type 1's (group 5), multiple type 2's (group 6), and multiple type 3's (group 7).

We have monitored executions and memory performance of many groups of multiple interacting programs. Aiming at providing insights on the LRU page replacement behavior during program interactions, we select five representative program interaction groups to discuss in this paper. The performance re-



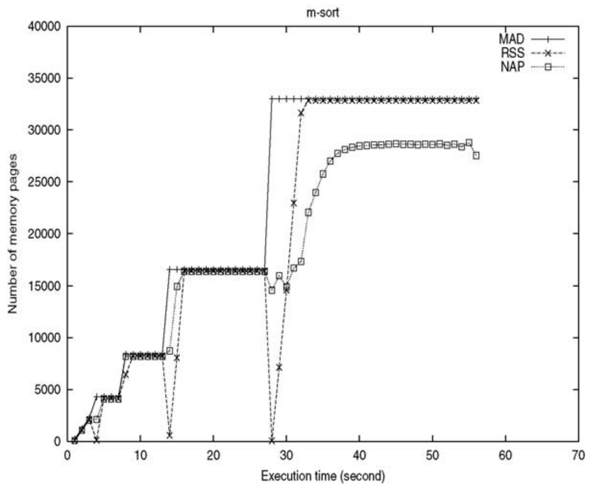


Fig. 4. The memory performance of m-m (left) and m-sort (right) in a dedicated environment.

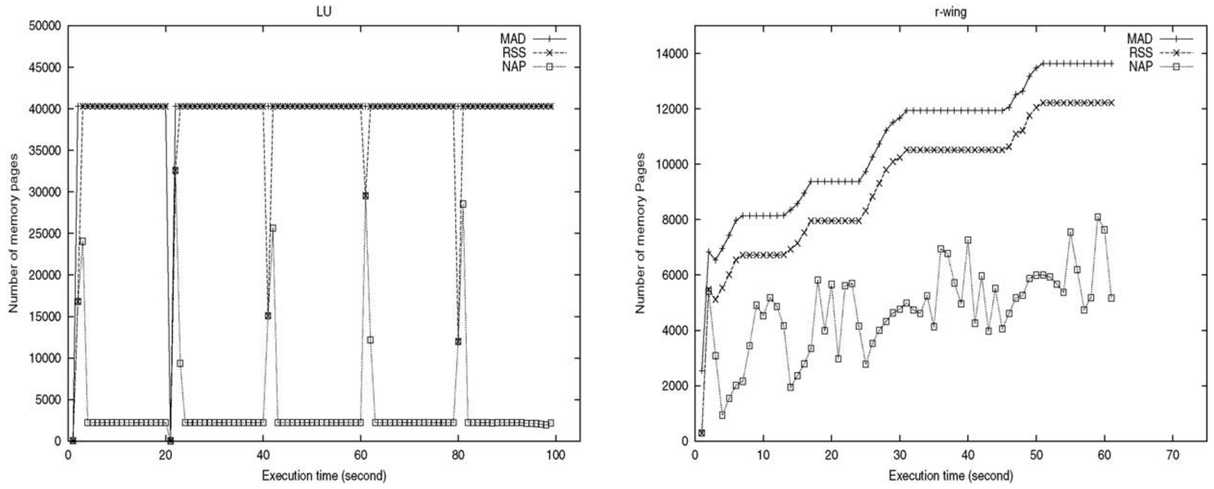


Fig. 5. The memory performance of LU (left) and r-wing (right) in a dedicated environment.

sults of many other program interactions are consistent with the reported ones. In order to clearly and concisely present effects of the false LRU pages on the program executions, we selected two programs in each group. Our experiments show that this is the best case revealing the insights of false LRU pages without the involvement of load controls.

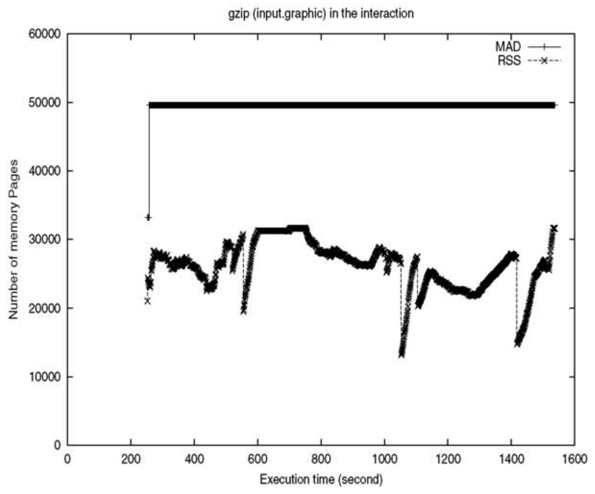
The five selected program interaction groups are *gzip* interacting with it vortex3 (belonging to group 1)*, bit-r* interacting with *gcc* (belonging to group 2), *vortex3* interacting with *gcc* (belonging to group 3), *vortex1* interacting with *vortex3* (belonging to group 6), and *LU-1* interacting with *LU-2*[4] (belonging to group 7).

The available user memory space was adjusted by the monitoring program accordingly so that each interacting program had considerable performance degradation due to 20–50% memory shortage.[5] As the program execution reaches the shortage range, these memory-constrained programs start thrashing, but are not completely page-fault I/O bound. It is the range where improvements to page replacement algorithms can help the most. Our work aims at eliminating thrashing in this situation and leave the true page-fault I/O bound situation to load controls.

Fig. 6 presents the memory usage behaviors measured by MAD and RSS of interacting programs *gzip* (left) and *vortex* (right). In the figures, both RSS curves fluctuate during the interacting execution, which shows the conflicts between memory demands and the limited memory allocations for each program exist for the long period of time, even though the memory is enough to satisfy the need for one program at a time. A program gains pages and increases its RSS through page faults. Meanwhile, it loses pages when these pages become old. In this way global page replacement policy tries to make the memory allocated among multiple programs conform their memory needs. Unfortunately, what a program loses includes false LRU pages, which are generated during its period of faulting. The losing of these false LRU pages does not reflect the memory needs. Our study shows that the proportion of false LRU pages among all the

[4] LU-1, LU-2 are two executions of LU with the same input parameters.

[5] The shortage ratios are calculated based on the peak memory demands during program executions. In practice, the realistic memory shortage ratios are smaller due to dynamically changing memory demands of interacting programs.



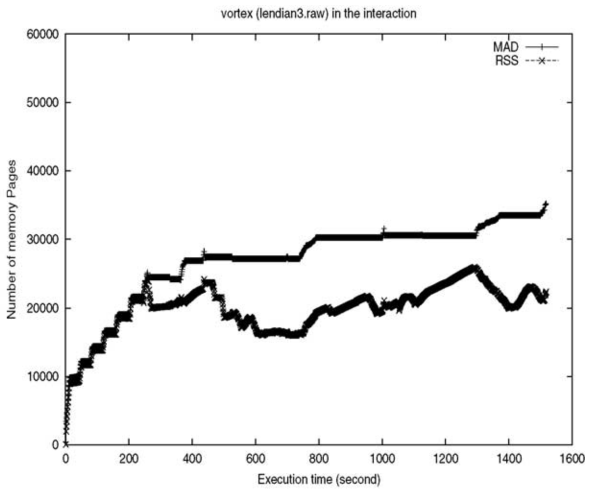


Fig. 6. The memory performance of gzip (left) and vortex3 (right) during the interactions.

page faults keeps increasing with the increase of memory shortage. Consequently, the dynamic memory allocations do not reflect the memory needs of programs. For example, *gzip* established its working set during the period of time between 600th second and 760th second, because we observed that its page fault rate is significantly lowered. Then some of its memory allocation was shifted to *vortex*, illustrated by the lowered *gzip* RSS curve and increased *vortex* RSS curve after 760th second in Fig. 6. We believe the pages *gzip* lost is part of its working set, because it had much more page faults and tried to gain some allocation back after then. Though *vortex* can “snatch” certain memory spaces from *gzip*, it is unable to build up its working set. This is because it also loses a large number of false LRU pages in its working set build-up process, which should not have been lost considering the needs of *vortex*. Unfortunately, we observed that the system ended up with large page fault rates for both programs and low CPU utilization. We found that a program is powerful to get additional memory allocation in the global replacement policy when it has large memory shortage between its RSS and its working set. However, when it gets more memory, it becomes less powerful, and is prone to lose memory. For this reason we see the fluctuating RSS curves on the interacting programs during their thrashings. Our experiments show that the execution times of both programs are significantly increased due to the page faults in the interaction. The slowdown of *gzip* is 5.23, and is 3.85 for *vortex*.

Fig. 7 presents the memory usage behavior measured by MAD and RSS of interacting programs *bit-r* (left) and *gcc* (right). Program *gcc* belongs to type 3 which has two spikes in MAD and RSS due to its dynamic memory allocation demands and accesses (see the right figure in Fig. 1). For *bit-r*, the RSS curve dropped sharply at the 165th second caused by the first RSS spike of *gcc* at the same time. When the second RSS spike of program *gcc* arrived at the 365th second, the RSS of *bit-r* dropped again. However, this time the RSS of *gcc* began to lose its pages at about 450th second before it could establish its working set. After that, both programs exhibited fluctuating RSS curves. The second spike requires only 7% more memory demand than the first spike, which makes a much longer execution delay. Our experiments consistently show that the execution times of both programs were significantly increased due to the page faults in the interaction. The slowdown of *bit-r* is 2.69, and is 3.63 for *gcc*.

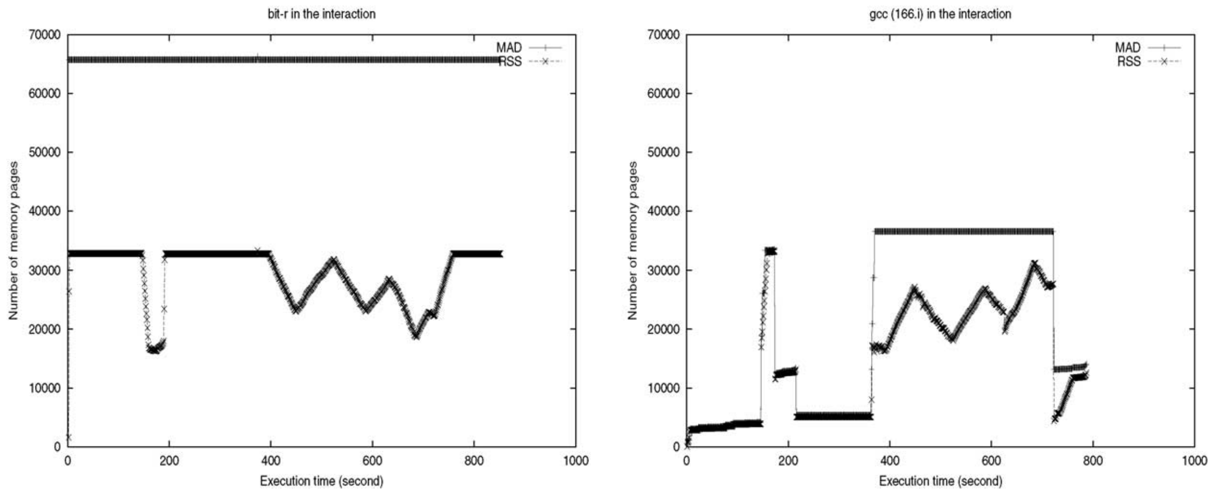


Fig. 7. The memory performance of bit-r (left) and gcc (right) during the interactions.

Fig. 8 presents the memory usage behavior measured by MAD and RSS of interacting programs *gcc* (left) and *vortex3* (right). The fluctuating RSS curves of the *vortex3* and the first spike of *gcc* caused a large number of page faults to each program, which delayed the first spike of *gcc* by 865 s, and delayed a RSS stair in *vortex* by 563 s. The second spike of *gcc* arrived after *vortex3* finished its execution, so it went smoothly. Our experiments consistently show that the execution times of both programs were significantly increased due to the page faults in the interaction. The slowdown of *gcc* is 5.61, and is 3.37 for *vortex*.

Fig. 9 presents the memory usage behavior measured by MAD and RSS of interacting programs *vortex1* (left) and *vortex3* (right). Although the input files are different, their memory access patterns of the two programs are the same. The RSS curves of both *vortex* programs changed similarly during the interactions.

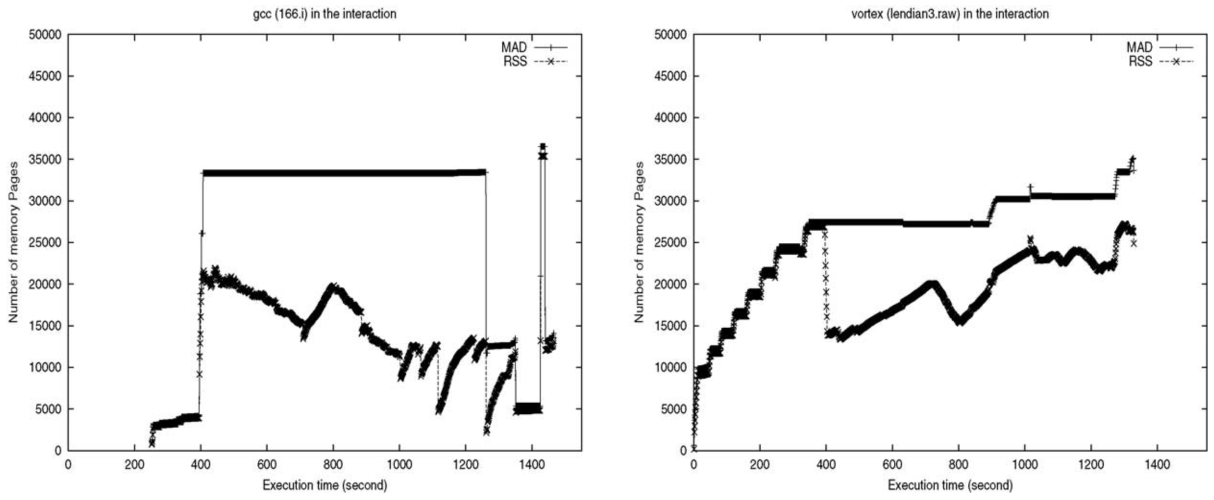
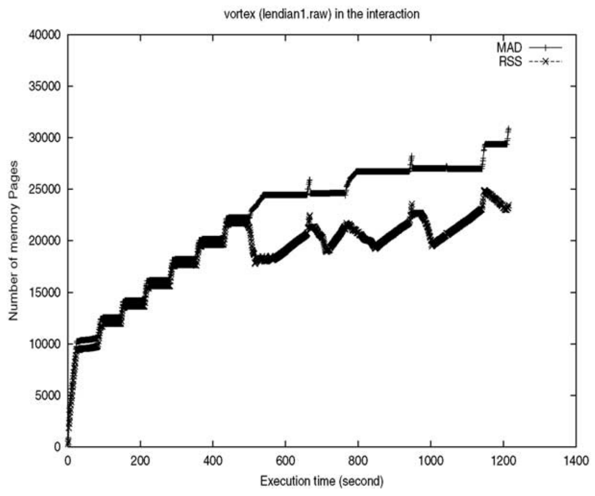


Fig. 8. The memory performance of gcc (left) and vortex3 (right) during the interactions.



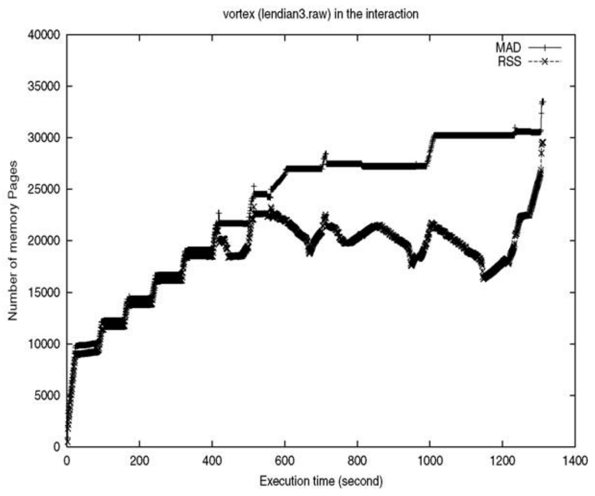
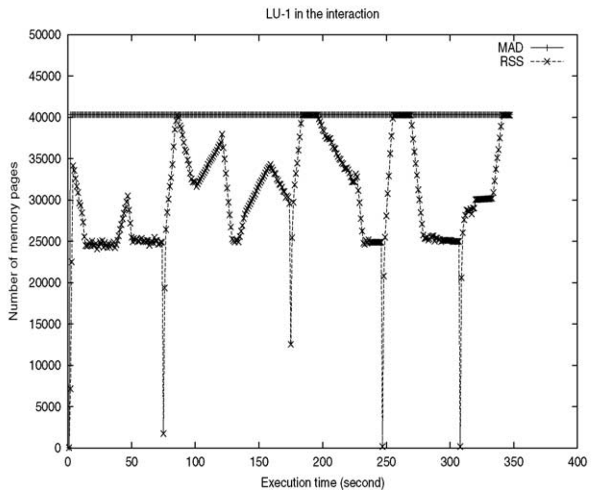


Fig. 9. The memory performance of vortex1 (left) and vortex3 (right) during the interactions.

But neither could establish its working set. 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 is 3.33 for *vortex3*.

Fig. 10 presents the memory usage behavior measured by MAD and RSS of two interacting programs *LU*. Our experiments show that frequent climbing slopes of RSS can incur memory reallocations and trigger fluctuating RSS curves, leading to inefficient memory usage and low CPU utilization. The dynamic memory demands from the programs caused the system to stay in the thrashing state for most of the time. The execution times of both programs were significantly increased due to the page faults in the interaction. The slowdown for *LU-1* is 3.57, and is 3.40 for *LU-2*.



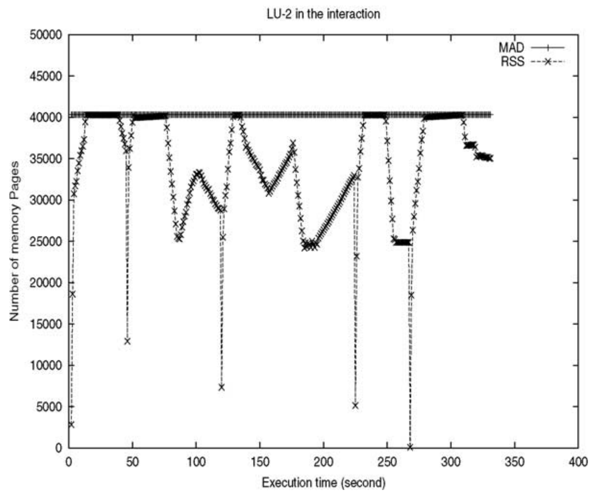


Fig. 10. The memory performance of LU-1 (left) and LU-2 (right) during the interactions.

4.3. How are thrashings triggered?

We have experimentally shown thrashings can be triggered with a moderate memory shortage and cause significant performance degradations. False LRU pages play their role in the process—they make global replacement policies blind to the program actual memory needs, while a portion of the working set identified as the false LRU pages are mistakenly replaced. Here are certain conditions that probably cause thrashings based on our experimental studies.

- When the memory demand of a program has a sudden jump for additional memory allocation, its RSS can be easily increased accordingly in the beginning because the newly added pages do not need loading pages through I/O (zero-filled pages rather than disk-read pages). If the program can not establish its working set before many false LRU pages are generated, the number of lost pages on the false LRU condition will exceed the number of obtained pages through page faulting, causing its RSS to drop. In addition, the increased memory demand of this program causes other programs in the system to generate more false LRU pages. In this way thrashings are generated. The starting execution stage of *gzip* in the left figure of Fig. 6, the second spike of *gcc* in the *bitr*/*gcc* interaction in Fig. 7, the first spike of *gcc* in the *gcc*/*vortex* interaction in Fig. 8, and all the RSS jumps of both *LU* programs in the *LU1*/*LU2* interaction in Fig. 10 are the examples of this condition.
- If memory access patterns of interacting programs in terms of working set size, memory usage behavior, and access frequency are similar, false LRU pages can be easily generated for both programs to cause a thrashing in the system. The *vortex*/*vortex* interaction and *LU1*/*LU2* in Figs. 9 and 10 (ples of this condition.
- When the available memory space is significantly less than the total memory demands of the interacting programs, all the programs compete for the limited memory allocations. A small number of page faults can easily trigger the process of generating a large number of false LRU pages. This condition will be shown in both the left and right figures of Fig. 11 in Section 5.4 before the token is taken by a program. (We will explain this figure in detail soon.)

5. Design and implementations of the token-ordered LRU

We choose Linux OS as a base to evaluate our design and implementation of the token-ordered LRU. We use Kernel 2.2 [3] as an original Linux system to demonstrate the effectiveness of our scheme.

5.1. LRU page replacement in Linux

An approximated LRU policy is implemented in the Linux kernel for page replacement. When a page fault occurs, kernel function “do_page_fault()” will be called to handle it. If the page fault is caused by a legal access to a page missing in memory but stored in the swap file in disk, the kernel will try to get a free memory page and load the requested page from the swap file by the kernel function “do_swap_page()”. If there are no free memory pages available, the kernel will make a room for the page by selecting a page from physical memory for replacement. If the replaced pages are dirty, they have to be saved in the swap file first, which also contributes to the number of major page faults (NPF).

Kernel function “__get_free_pages()” will be invoked by the swap daemon (kswapd), which will be waken up to function when the free physical memory space becomes scarce, or when a page faulted program can not find a page from the free page pool (“free_area”). The function will look at each possible program in the system to see if it is a candidate from which memory pages can be selected for swapping. It always starts from the program with the largest resident pages. The kernel will then check through all of the virtual memory pages in the page table of the selected program. Principally, once the kernel detects that the reference bit of a page table entry is off (indicating that the page has not been accessed since it was reset last time by the function), the kernel will swap out the page. If the bit is on, the kernel will turn it off, and then check the next page in the table. If no pages can be paged out from this program, the next candidate program will be tried. This simple implementation effectively simulates the behavior of LRU replacement with a small overhead. However, it also generates false LRU pages during program interactions as we have discussed in the previous sections.

Operating systems have protection mechanisms to resolve serious thrashing problems. For example, a process will be removed to release its memory space when a thrashing occurs in Linux. A process will be swapped out for the same purpose in 4.4 BSD operating system. If the free page pool can not be filled timely, the system will start to swap out or remove processes.

Unfortunately, the existence of false LRU pages makes kernel function “__get_free_pages()” in Linux (and the “pageout” daemon in 4.4 BSD operating system) easily and quickly find “qualified” pages including many false LRU pages to fill the free page pool. As the result, the system can be involved in a “pre-thrashing” state for a significantly long time before the kernel is awakened to swap out or remove processes. Our experiments will show that CPU utilization in the pre-thrashing state can be extremely low due to the large number of page faults. The system developers of the 4.4 BSD operating system indicates that system performance can be much better when the memory scheduling is done by page replacement operations than when the process swapping is used [22]. Our token-ordered LRU is a page replacement oriented memory scheduling scheme to address the thrashing problem before the system has to swap out or remove processes.

5.2. *How is the token-ordered LRU implemented in Linux?*

The basic idea of the token-ordered LRU is to block the spread of false LRU pages among all the interacting programs, and to make the working set of at least one program be identified and established.

A token is a newly introduced global and mutually exclusive variable in the kernel, which has two states: 1 means that the token is available, 0 means that the token has been taken by a page faulted program. The token is initialized when the system is booted. In our implementation, we make a program request the token before invoking function “do_swap_page()”, which is right after a page fault occurs and before the page will be loaded from the swap file to ensure the token only goes to the program in need. The token is only taken by a program when page faults occur due to memory shortage. In other words, a program will not compete for the token until the memory space is insufficient for it. The system functions exactly as the original Linux system when memory space is sufficient for programs. After the token is taken by a program, its status will be reset to 0.

We have also added a new field, called swapping_status, for each program in its *task_struct* data structure, which is turned on when the program is in the stage of swapping in/out pages. It will be turned off when the program returns to its normal execution stage from page faults.

As we have discussed, the false LRU pages can be generated in a program during its page swapping period. The false LRU pages are avoided for a program holding the token as follows. During the process of selecting and marking LRU pages (by turning off the reference-bits) for page replacement, kernel function "__get_free_pages()" skips the program holding both the token and the swapping status. In this way the memory pages of the program with the token are protected when and only when it has unsolved page faults, and false LRU pages are eliminated from it.

The selected and marked LRU pages of a program during a normal computing phase are the true LRU pages, which are also the replacement candidates targeted by our token-ordered policy. To address this, we invalidate the privilege for the program holding the token as soon as the program resolves its page faults by turning off its swapping status. The kernel function "__get_free_pages()" will then include the program for LRU page searching.

In our implementation, we have an exception handler. When the privileged process could not find LRU pages from other programs for replacement, the system will have to select the LRU pages for the program from its own resident space.

The only additional operations for the token-ordered LRU are cycles used to set the token/swapping status, and to decide if the a program holding the token will be skipped or not in the process of selecting and marking LRU pages. These cycles are negligible. Thus, the token implementation causes very little overhead.

5.3. Objectively monitoring the usage of the token

The highest priority on the token assignment in a parallel or distributed system is given to the process with the synchronization or other blocking messages waiting on it, because relieving the process from thrashing benefits not only the process itself but also other processes coordinating with it. This is especially important for high performance computing (HPC) applications. A large portion of them are modeled using the bulk-synchronous parallel (BSP) model, in which the entire application must wait for the slowest process before it can synchronize [11].

If no processes involving in the thrashing belong to this category, we initially allow the token to be taken by the interacting programs in a random order. We hope a program holding the token effectively utilizes the memory system, and finishes the execution as soon as possible to release the space and the token for other interacting programs. However, a program holding the token may abuse the privilege and significantly degrade the performance of other programs in following two cases. First, when the memory allocation demand of the program is larger than the total user memory space, the program itself will cause page faults even if all the memory space is reserved for it. At the same time, other interacting programs are suffering from system thrashing. We assume that the memory demand of a program is unknown in advance, and this situation can only be detected at runtime. Second, if the execution time of the program is significantly longer than other interacting programs, the token assignment is not fair to other programs. In fact, as we know, the optimal scheduling strategy is based on a principle of shortest-job-first [6]. We again assume that the lifetime of any program is unknown in advance.

During program interactions, the token-ordered LRU system is monitoring the activities of each program. The first case is detected if we find that the program holding the token frequently selects pages from itself during the page faults (no other programs can provide LRU pages to it). This program is and would be experiencing a large amount of page faults without necessary actions. We will then force the program to return the token, and mark the program as "token-unsuitable".

The second case is detected if the execution time of the program holding the token exceeds a pre-determined threshold. We will relinquish its token and mark the program as “token-unsuitable”. This label will be taken off after another pre-determined time period to ensure that the program is treated fairly. However, frequent token transferal among the interacting programs will cause considerable page faults. Thus, the pre-determined threshold should be set large enough to avoid this scenario.

5.4. A close look at the token-ordered LRU in program interactions

To show how a token functions and its effectiveness, let’s have a close look at its running behavior during program interactions. The following program segment is used in the experiment:

```
#define LOOP 1000
  double * mem_page;

  mem_page = (double *)calloc(SIZE, sizeof(double));
  for (i = 0; i < LOOP; i++){
    for (j = 0; j < SIZE; j += step){
      mem_page[j] = mem_page[j] + 1;
      Other computing work only on mem_page[j];
    {
    if ( (i+1)%10 == 0 )
      SIZE = (long)(0.9*SIZE);
  }
```

This program consists of 1000 loops to access a large allocated array. At first the program sequentially accesses its entire data array 10 times. Then for each of its next 10 loops, the program reduces its accessing scope of the data array by cutting 10% of all its accesses at the end of the array. The available user memory space was adjusted to 60 MB. The cyclic access pattern produces a large number of page faults when there is a memory shortage, which has been observed and studied in recent papers [12,16,25] in a dedicated environment. In this example, we will show how the token works to address the serious performance degradation by reducing false LRU pages in a practical program interaction environment.

We let two instances of the program run simultaneously, allocating a 53 MB array for one process (referred as small process in the following) and a 58 MB array for another process (referred as large process in the following) by adjusting variable SIZE in the program segment. Closely tracing the page access behaviors of each program before and after the token was set in the system, we present the impact of the token to each of the interacting programs. Fig. 11 presents space-time graphs for the small process (left) and the process (right) during their interaction, where *y*-axis represents three types of memory pages at different virtual addresses: recently visited pages,[6] swapped-out pages, and resident but not recently visited pages, and the *x*-axis represents the execution time sequence. The RSS size of each process can be approximated by the sum of the number of “visited pages” and the number of “resident but not visited pages” at any execution point. We have observed that each of the processes expanded its

[6] *Recently* refers to the previous 1 s.

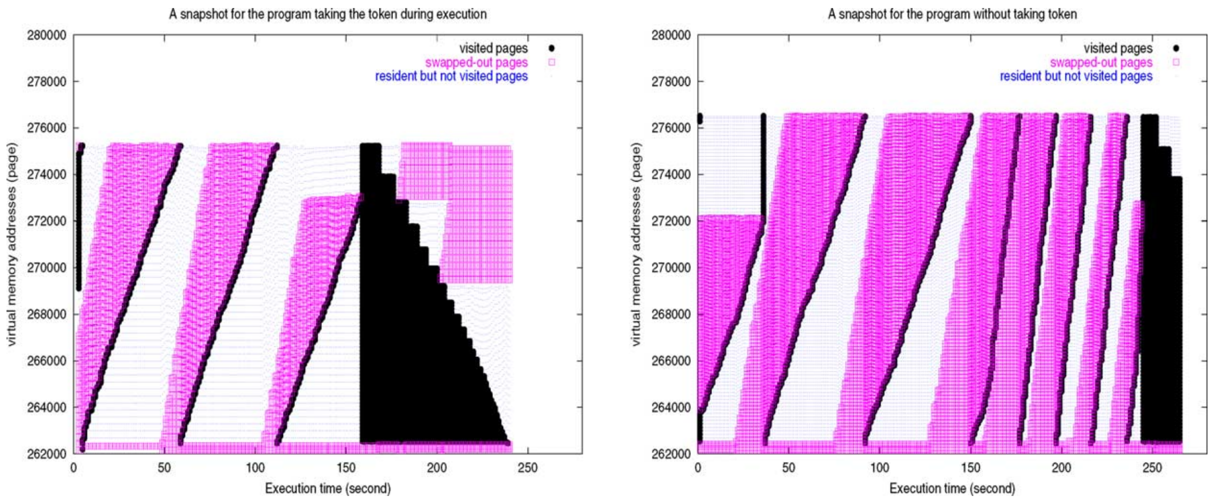


Fig. 11. The memory behaviors of the data access process with 53 MB data array, which took the token in the middle of the execution (left) and the other process with 58 MB data array, which did not take the token (right) during their execution interaction.

RSS through page faulting while some of its pages were replaced under the false LRU condition in the process. The combination of these two activities caused three consequences: (1) neither process could establish its working set; (2) the RSS size of each process fluctuated; and (3) little useful work could be done.

The token was set in the system and taken by the small process (left figure in Fig. 11) at the execution time of 125th second. After that time, this process successfully kept its useful memory pages and avoided false LRU pages, whose effect was reflected by the increased lightly gray area of "resident but not visited pages" in the left figure of Fig. 11. During the same period of time, the large process reduces the number of its "resident but not visited pages" (see right figure in Fig. 11). Once the small process establishes its working set, the left figure in Fig. 11 shows that all its obtained pages are frequently visited. The token only avoids swapping out the false LRU pages, and still treats the true LRU pages as replacement candidates. This has been confirmed by observing the phase when the small process starts reducing its working set. Although the process still held the token, its true LRU pages were migrated to the large process so that the big process can utilize these released memory pages. The right figure in Fig. 11 shows that the large process increased its RSS size at that time. The large process quickly finished its execution after the small process holding the token left the system.

It is interesting to see that the token is also beneficial to the process that did not take the token. The right figure in Fig. 11 shows that the large process without the token took about 50 s to finish one pass of accesses to the data array before the token was set in the system. After token was taken by the small process, one pass access time was reduced to less than 25 s, although its RSS was reduced. The reason for this is as follows. Since I/O bandwidth to/from disk becomes a bottleneck when a system conducts a large number of page faults for both processes, the page fault penalty increases accordingly. When one process gets the token, its number of page faults are significantly reduced, and it consumed much less I/O bandwidth. Thus, the page fault penalty of the process without the token is greatly reduced by highly utilizing the available I/O bandwidth, and more useful work can be done even with additional page faults.

6. Performance of the token-ordered LRU

The performance of the token-ordered LRU is experimentally evaluated by the five selected groups of the interacting programs. Each of the experiments has the exactly same condition as its counterpart conducted in Section 4.2, except that the page replacement of each group of interacting programs is managed by the token-ordered LRU.

Fig. 12 presents the memory performance measured by MAD and RSS of interacting programs *gzip* (left) and *vortex* (right) in the token-ordered LRU environment. During the interactions at the execution time of 250th second, both programs started page faults due to a memory shortage. The token was taken by program *vortex* after then. Fig. 12 shows that the once seriously fluctuating RSS curves of *vortex* observed in the original system in Fig. 6 disappeared. Although its RSS did not exhibit the behavior as it is shown in the dedicated environment, where its RSS curve was almost overlapped with its MAD curve (see the *vortex* graph in Fig. 3), we believe the RSS curve represents its necessary memory allocation demands for its effective execution (or its working set size). There are two reasons for this: (1) The page fault rate is significantly lower than that in its counterpart experiment for the original system. Even when RSS curve of *vortex* is considerably lower than its MAD curve after the 470th second, its page fault rates are lowered by at least 70% compared with those measured at the same execution stage in the original system. (2) The RSS curve of *vortex* is consistent with its NAP curve in the dedicated environment (also see the *vortex* graph in Fig. 3). The NAP curve increased much slowly than MAD curve, which reflects the recently accessed memory size did not increase as MAD did. So the gap between its RSS and MAD curves in Fig. 12 was enlarged in its late execution stage, where its fluctuation was caused by the content change of its working set. While eliminating the thrashing quickly, the token-ordered LRU distinguishes true and false LRU pages, and only keeps the working set of the protected process in the memory, rather than simply pins all of its pages in memory. Our experiments also show that the execution times of both programs were significantly reduced by the token-order LRU algorithm compared with the times with the original Linux LRU. The slowdown of program gzip is 2.63 (a reduction of 50%), and is 1.83 for

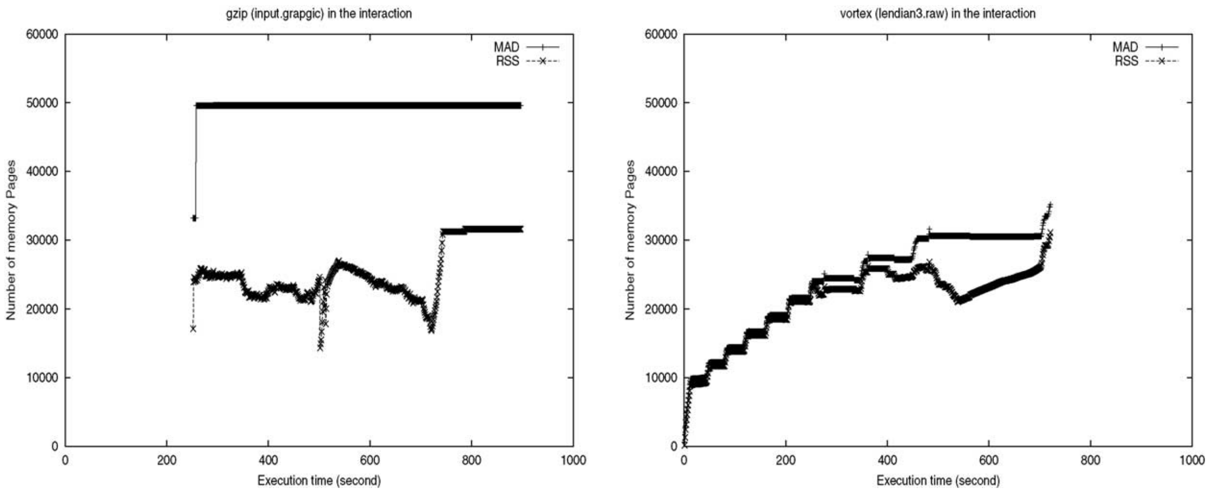


Fig. 12. The memory performance of gzip (left) and vortex (right) during the interactions managed by the token-ordered LRU.

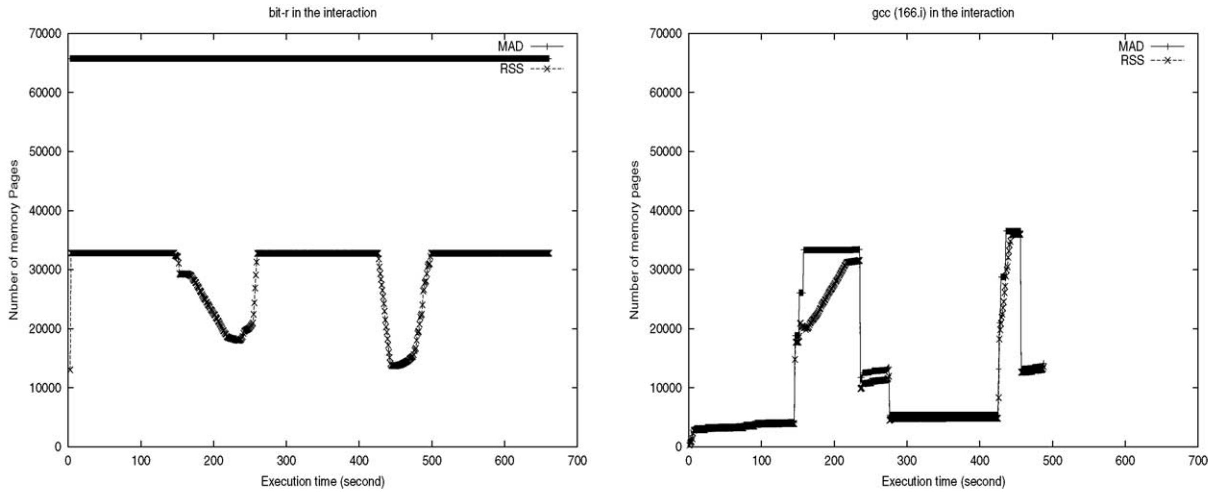


Fig. 13. The memory performance of bit-r (left) and gcc (right) during the interactions managed by the token-ordered LRU.

program vortex (a reduction of 52%). The page fault reductions for programs gzip and vortex are 45% and 80%, respectively.

Fig. 13 presents the memory performance measured by MAD and RSS of interacting programs *bit-r* (left) and *gcc* (right) in the token-ordered LRU environment. During the interactions at the execution time of 146th second, the first RSS spike of program gcc caused page faults of both programs due to memory shortage. The token was taken by *gcc* after that point. Fig. 13 shows that *gcc* quickly built up its working set, reflected by keeping its first RSS spike with a short delay after taking the token (right), while program *bit-r* sharply decreased its RSS during this short period of time. Program *gcc* established its working set in its second spike more quickly than it did in its first spike, due to the difference between their reference behavior: *gcc* accesses its working set more frequent in the second spike than it does in the first spike. The token-ordered LRU attempts to reduce false LRU pages without affecting the ability of global LRU to reflect memory access patterns of running programs. Our measurements show that the execution times of both programs were significantly reduced by the token-order LRU compared with the times from the original Linux LRU. The slowdown of *bit-r* is 2.08 (a reduction of 23%), and is 2.25 for *gcc* (a reduction of 38%). The page fault reductions for *bit-r* and *gcc* are 20% and 82%, respectively.

Fig. 14 presents the memory performance measured by MAD and RSS of interacting programs *gcc* (left) and *vortex3* (right) in the token-ordered LRU environment. During the interactions at the execution time of 397th second, both programs started page faults due to memory shortage. The token was taken by *gcc* after that time. Fig. 14 shows that *gcc* quickly built up its working set, reflected by keeping the first RSS spike narrow after taking the token (left), while *gzip* sharply decreased its RSS during this short period of time. Program vortex3 finished before the second RSS spike of program gcc arrived. Then *gcc* finished its execution without major page faults after 42 s. Our measurements also show that the execution times of both programs were significantly reduced by the token-order LRU compared with the ones with the original Linux LRU. The slowdown of *gcc* is 1.85 (a reduction of 67%), and is 1.54 for *vortex* (a reduction of 54%). The page fault reductions for *gcc* and *vortex* are 95% and 79%, respectively.

Fig. 15 presents the memory performance measured by MAD and RSS of interacting programs *vortex1* (left figure) and *vortex3* (right) in the token-ordered LRU environment. During the interactions at the

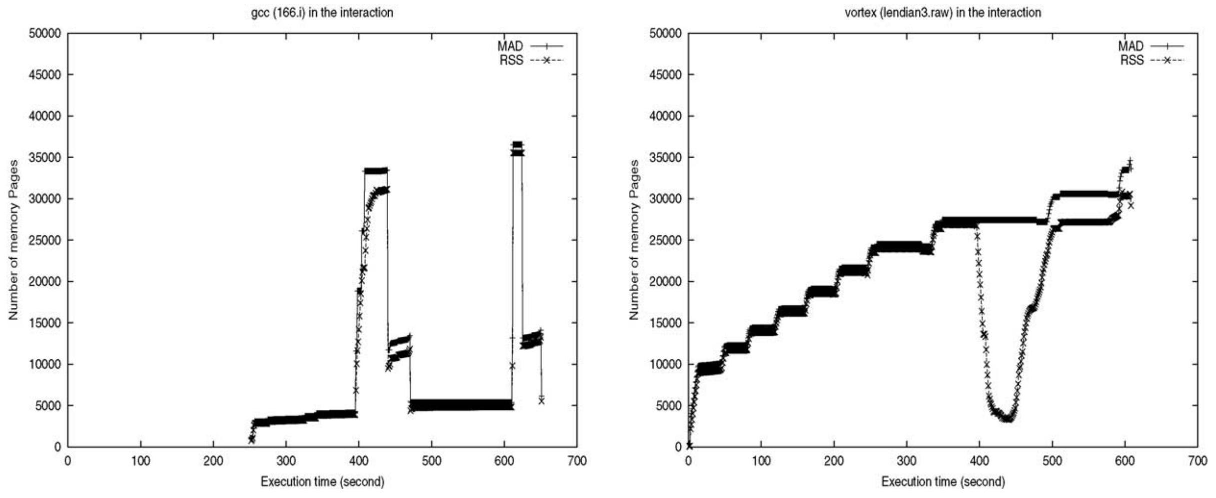


Fig. 14. The memory performance of gcc (left) and vortex3 (right) during the interactions managed by the token-ordered LRU.

execution time of 433th second, both programs started page faults due to memory shortage. The token was taken by *vortex1* after that point. Fig. 15 shows that the program quickly built up its working set, reflected by its climbing RSS curve after the token was taken (left). Our measurements also show that the execution times of both programs were significantly reduced by the token-order LRU compared with the ones with the original Linux LRU. The slowdown of *vortex1* is 1.95 (a reduction of 46%), and is 2.08 for *vortex3* (a reduction of 38%). The page fault reductions for *vortex1* and *vortex3* are 93% and 63%, respectively.

Fig. 16 presents the memory performance measured by MAD and RSS of interacting programs *LU-1* (left) and *LU-2* (right) in the token-ordered LRU environment. In the first spikes of both *LU-1* and *LU-2*

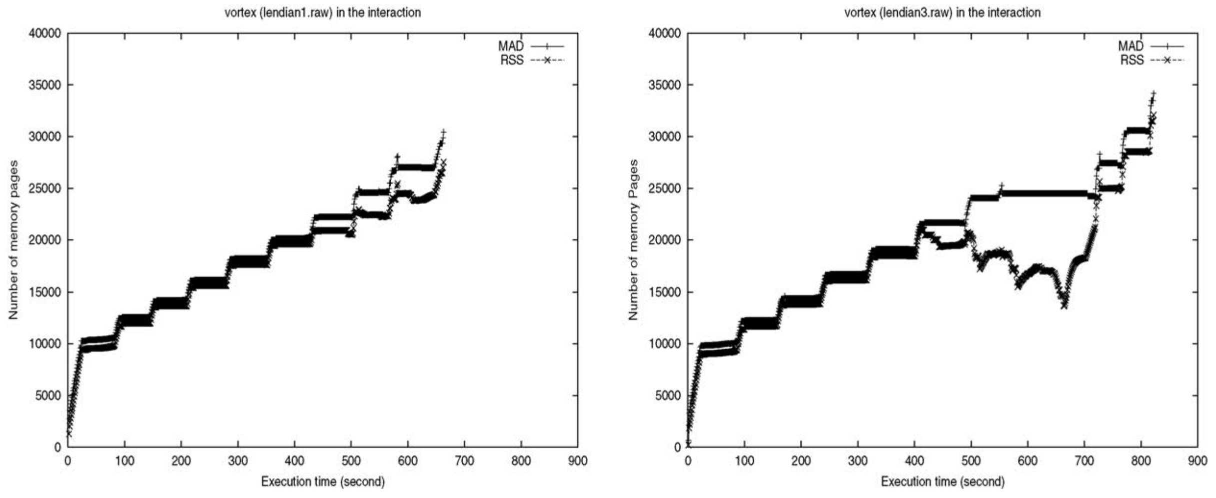
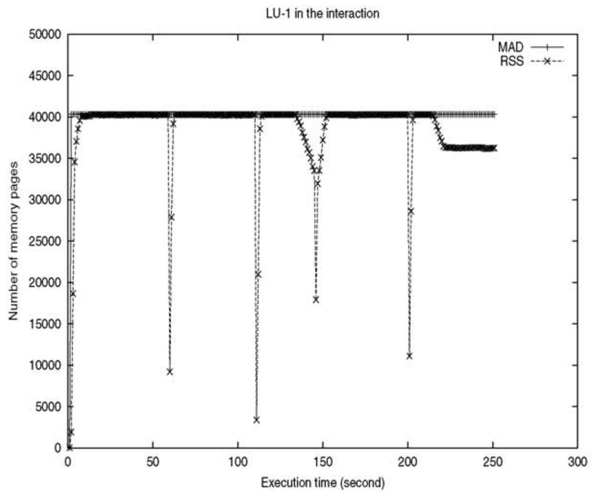


Fig. 15. The memory performance of vortex1 (left) and vortex3 (right) during the interactions managed by the token-ordered LRU.



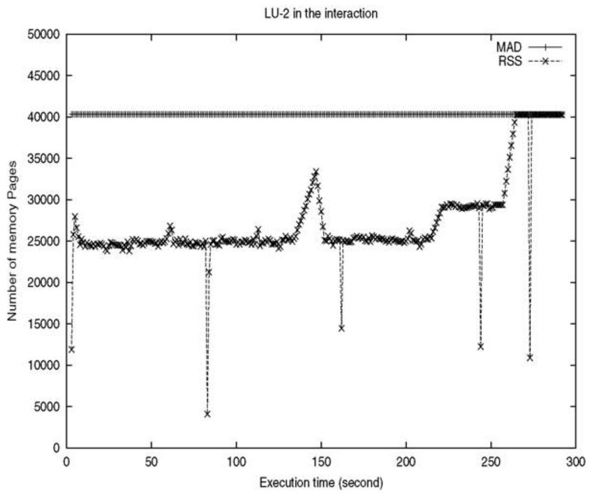


Fig. 16. The memory performance of LU-1 (left) and LU-2 (right figure) during the interactions managed by the token-ordered LRU.

programs after a few seconds of executions, both programs started page faults due to memory shortage. The token was taken by *LU-1* after that point. Fig. 16 shows that *LU-1* quickly built up its working set, reflected by keeping its RSS curve very similar to its RSS curve in the dedicated environment in the left figure of Fig. 5 after taking the token (the left figure), while *LU-2* could only obtain a moderate level of RSS during this period of time. In the last 25 s of the execution of *LU-1*, its RSS curve was lowered while the RSS curve of *LU-2* adaptively rose by obtaining true LRU pages from *LU-1*. Our measurements also show that the execution times of both programs were significantly reduced by the token-order LRU compared with the ones with the original Linux LRU. The slowdown of *LU-1* is 2.57 (a reduction of 28%), and is 2.99 for *LU-2* (a reduction of 12%). The page fault reductions for *LU-1* and *LU-2* are 87% and −116%, respectively. It is noted that the execution time was still reduced, though the number of page faults of *LU-2* increased. This is because the page fault penalty decreased with more available I/O bandwidth after the token was taken by *LU-1*.

In summary, we list the performance improvements of the token-ordered LRU in terms of the reduction of page faults and execution slowdowns in Table 2. In addition, unlike the suspension mechanism in the

Table 2
Comparisons of the total number of page faults and slowdowns of each program during the interactions managed by LRU and the token-ordered LRU, where the page fault reduction represents the reduced percentage by the token-ordered LRU over the normal LRU

	gzip/vortex	bit-r/gcc	vortex/gcc	vortex-1/vortex-3	LU-1/LU-2
LRU page faults	328,551/294,038	108,352/139,807	248,045/275,982	193,399/208,342	138,372/77,680
Token LRU page faults	179,908/59,945	86,467/25,406	51,749/12,425	13,842/76,171	17,999/168,215
Page fault reduction	45%/80%	20%/82%	79%/95%	93%/63%	87%/(-117)%
LRU slowdown	5.23/3.85	2.69/3.63	3.37/5.61	3.58/3.33	3.57/3.40
Token LRU slowdown	2.63/1.83	2.08/2.25	1.54/1.85	1.95/2.08	2.57/2.99
Slowdown reduction	50%/52%	23%/38%	54%/67%	46%/38%	28%/12%

load control, the process(es) without token still can keep their memory spaces not used by the token possessed process and continue their computing. This is especially helpful for processes involved in coordinated computing across multiple nodes.

7. Other related work

Management of memory hierarchies has been a topic of study for several decades. Regarding the large gap in access time between memory and disk, a lot of work has been done to reduce the number of page faults for each program. Page replacement algorithms have been a classical topic since 1960s (see e.g. [1,4] for early work). LRU page replacement policies have been recently studied in dedicated environment through intensive simulations(see e.g. [12,16,25]).

Regarding program interactions, process scheduling has been a focused topic (e.g. [10]). Regarding kernel development for memory performance improvement, researchers have tried to link users at the application level to the kernel so that kernel is well informed for page replacement (see e.g. [13,23]). Although the above cited work has improved memory performance by focusing on reducing page faults, the page replacement policies for program interactions have not been the main focus. A recent work to address thrashing protection in the multiprogramming environment was conducted by Jiang and Zhang [17]. They proposed a facility called TPF in Linux kernel to monitor the program interaction. Once a thrashing is detected, a program is protected by preventing its pages from eviction. Token-based LRU has three major differences from the TPF facility: (1) TPF is a detection-based, reactive mechanism. However, token-based LRU is a proactive scheme. Thus it can provide more steady protection. (2) While TPF pins all of the pages of a protected program in memory, the token-based LRU discriminates false LRU pages from true LRU pages, and only prevents true LRU pages from eviction. In this way, the token-based LRU makes memory better utilized. (3) The token-based LRU does not contain the pre-defined parameters used in TPF to detect the existence of thrashings.

8. Conclusion

We have investigated sources of memory performance degradation in program interactions by carefully examining the LRU memory page replacement and its representative implementations in Linux systems. We have experimentally demonstrated that the false LRU pages can be a serious loophole in LRU replacement implementations because these implementations do not correctly predict and reflect memory access patterns of interacting programs.

In order to address the limitation in the LRU replacement in program interactions, we have proposed the token-ordered LRU policy, and implemented the policy in the memory management system of the Linux kernel. The experiments show that the token-ordered LRU algorithm consistently and significantly reduces the page faults and the execution slowdown of program execution in a multiprogramming environment. While thrashing is largely prevented at individual computing node in a parallel system using the token-ordered LRU, our technique would be of great help to the stability and performance of the whole system.

The design and implementation of token-ordered LRU can be applied in other resource management systems for job interactions, such as job scheduling in distributed systems. For example, in a cluster of servers, an owner user may be willing to share the resource with remote users who submit or migrate their jobs to the owner node. However, thrashing caused by memory competition from remote jobs could seriously slowdown the owner jobs. Such risks may discourage owners from sharing their resources. However, by granting the owner's jobs with tokens, the owner users can prevent their jobs from thrashing even when they widely open the resources to remote memory-intensive jobs.

Acknowledgment

The authors are grateful to Phil Kearns for providing a kernel programming environment. They also thank Bill Bynum for reading the paper and for his suggestions. We thank the anonymous referees for their critical and constructive comments to this work. This work has been supported in part by the US National Science Foundation under grants CNS-0098055 and CCF-0129883.

References

[1] A.V. Aho, P.J. Denning, J.D. Ullman, Principles of optimal page replacement, J. ACM 18 (1) (1971) 80–93.
[2] A. Alderson, W.C. Lynch, B. Randell, Thrashing in a Multiprogrammed System, Operating Systems Techniques, Academic Press, London, 1972.
[3] M. Beck, et al., Linux Kernel Internals, second ed., Addison-Wesley, Reading, MA, 1998.
[4] E.G. Coffman, P.J. Denning, Operating Systems Theory, Prentice-Hall Englewood Cliffs, NJ, 1973.
[5] E.G. Coffman Jr., T.A. Ryan, A study of storage partitioning using a mathematical model of locality, Commun. ACM 15 (3) (1972) 185–190.
[6] R.W. Conway, W.L. Maxwell, L.W. Miller, Theory of Scheduling, Addison-Wesley, Reading, MA, 1967.
[7] P.J. Denning, The working set model for program behavior, Commun. ACM 11 (5) (1968) 323–333.
[8] P.J. Denning, Thrashing: its causes and prevention, in: Proceedings of AFIPS Conference, 1968, pp. 915–922.
[9] P.J. Denning, Virtual memory, Comput. Surv. 2 (3) (1970) 153–189.
[10] S. Evans, K. Clarke, D. Singleton, B. Smaalders, Optimizing unix resource scheduling for user interaction, in: Proceedings of the Usenix Summer 1993 Technical Conference, June 1993.
[11] E. Frachtenberg, D. Feitelson, F. Petrini, J. Fernandez, Flexible coscheduling: mitigating load imbalance and improving utilization of heterogeneous resources, in: Procedings of the International Parallel and Distributed Processing Symposium (IPDPS03), 2003.
[12] G. Glass, P. Cao, Adaptive page replacement based on memory reference behavior, in: Proceedings of 1997 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems, May 1997, pp. 115–126.
[13] K. Harty, D.R. Cheriton, Application-controlled physical memory using external page-cache management, in: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, pp. 187–197.
[14] HP Corporation, HP-UX 10.0 Memory Management White Paper, January 1995.
[15] IBM Corporation, AIX Versions 3.2 and 4 Performance Tuning Guide, April 1996.
[16] S. Jiang, X. Zhang, LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance, in: Proceedings of 2002 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems, 2002, pp. 31–42.
[17] S. Jiang, X. Zhang, TPF: a system thrashing protection facility, Software Pract. Experience 32 (3) (2002) 295–318.
[18] E.D. Lazowska, J.M. Kelsey, Notes on tuning VAX/VMS, Technical Report 78–12-01, Department of Computer Science, University of Washington, December 1978.
[19] J.B. Morris, Demand paging through utilization of working sets on the MANIAC II, Commun. ACM 15 (10) (1972) 867–872.
[20] L.J. Kenah, S.F. Bate, VAX/VMS Internals and Data Structures, Digital Press, Bedford, MA, 1984.

[21] K.-L. Ma, T.W. Crockett, A scalable, cell-projection volume rendering algorithm for 3D unstructured data, in: Proceedings of the Parallel Rendering'97 Symposium, October 1997, pp. 95–104.
[22] M.K. McKusick, K. Bostic, M.J. Karels, J.S. Quarterman, The Design and Implementation of the 4.4 BSD Operating System, Addison Wesley, 1996.
[23] D. McNamee, K. Armstrong, Extending the March external pager interface to accommodate user-level page replacement policies, in: Proceedings of USENIX March Symposium, 1990, pp. 17–29.
[24] NetLib, URL: http://www.netlib.org.
[25] Y. Smaragdakis, S. Kaplan, P. Wilson, EELRU: simple and effective adaptive page replacement, in: Proceedings of 1999 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems, May 1999, pp. 122–133.
[26] L. Xiao, X. Zhang, S.A. Kubricht, Improving memory performance of sorting algorithms, ACM J. Exp. Algorithmics 5 (2000).
[27] Z. Zhang, X. Zhang, Cache-optimal methods for bit-reversals, in: Proceedings of Supercomputing'99, 1999.

Song Jiang received the BS and MS degrees in computer science from the University of Science and Technology of China in 1993 and 1996, respectively, and received his PhD degree in computer science from the College of William and Mary in 2004. He is a postdoctoral research associate at the Los Alamos National Laboratory, developing next generation operating systems for high-end systems. He received the S. Park Graduate Research Award at the College of William and Mary in 2003. His research interests are in the areas of operating systems, computer architecture, and distributed systems.

Xiaodong Zhang received his BS degree in electrical engineering from Beijing Polytechnic University in 1982, MS and PhD degrees in computer science from University of Colorado at Boulder in 1985 and 1989, respectively. He is the Lettie Pate Evans Professor of computer science and the Department Chair at the College of William and Mary. He was the Program Director of Advanced Computational Research at the U.S. National Science Foundation from 2001 to 2003. He is a past editorial board member of *IEEE Transactions on Parallel and Distributed Systems*, and currently serves as an associate editor of *IEEE Micro*. His research interests are in the areas of parallel and distributed computing and systems, and computer architecture.