Advance Access published on February 15, 2006 doi:10.1093/comjnl/bxl001

Medium-Term Scheduler as a Solution for the Thrashing Effect†

MOSES REUVEN[1] AND YAIR WISEMAN[1,2,*]

[1]*Computer Science Department, Bar-Ilan University, Ramat-Gan, Israel*
[2]*School of Computer Science & Engineering, The Hebrew University of Jerusalem, Jerusalem, Israel*
[*]*Corresponding author: wiseman@cs.huji.ac.il*

We suggest a method for minimizing the paging on a system with a very heavy memory usage. Sometimes there are processes with active memory allocations that should be in the physical memory, but their total size exceeds the physical memory range. In such cases, the operating system starts swapping pages in and out of the memory on every context switch. We minimize this thrashing by splitting the processes into a number of bins, using Bin Packing approximation algorithms. We modify the scheduler to have two levels of scheduling—medium-term scheduling and short-term scheduling. The medium-term scheduler switches the bins in a Round-Robin manner, whereas the short-term scheduler runs the standard Linux scheduler among the processes in each bin. We show that this feature does not impose modifications on the shared memory maintenance. In addition, we show how to adjust the new scheduler to fit some elements of the original scheduler like priority and real-time privileges. Experimental results show significant improvement on heavily loaded memories. The code of this project is free and can be found in http://www.cs.biu.ac.il/~reubenm.

Keywords: Scheduling, allocation/deallocation strategies, swapping, virtual memory, process management

Received 27 October 2005; revised 16 January 2006

1. INTRODUCTION

One of the most important computer resources is the internal memory. A multitasking system executes multiple processes simultaneously. Each one of the processes uses part of the memory. The relationship between the memory and the scheduling strategy is an old issue for research [1].

Usually, most of the processes do not use the whole memory which is allocated to them. This leads to the idea of virtual memory [2]: many processes are stored in the virtual memory, but only the pages which are currently needed will be stored in the physical memory; hence many more processes can be executed concurrently, while using less physical memory.

Many operating systems implement the virtual memory scheme using the paging concept, i.e. the operating system loads a memory page into the physical memory only when a process demands it. If no free memory frame is available, the operating system swaps a page from the physical memory to the secondary memory (hard disk). Different methods for deciding which pages the operating system should swap out to the disk have been proposed over the years [3].

When too much memory space is needed, the CPU spends a large portion of its time swapping pages in and out of the memory. This effect is called thrashing [4]. Thrashing's result is a severe overhead time and as a result a significant slowdown of the system. Some studies for reducing the undesirable effects of the thrashing have been conducted over the years [5].

In [6, 7] the authors suggest giving one of the interactive processes the privilege of not swapping its page out. Consequently, the privileged process will succeed to be executed faster and will free its memory allocation earlier. This can help the operating system to clear out the memory and to return to a normal behaviour. However, such an approach will be beneficial only if the memory allocations slightly exceed the physical memory. This approach will act like a first-in-first-out (FIFO) scheduler when many processes generate a large memory excess. While this FIFO continues, the system will keep on thrashing. Linux 2.6 version has a similar mechanism and it is described in Section 2.

In [8] the authors suggest not to admit jobs which do not fit into the current available memory. Instead, the system waits for one or more processes to finish their execution. Only when enough memory is freed, a new job will be admitted.

†A preliminary version appeared in The Modelling, Simulation, and Optimization Conference, MSO-2005, Oranjestad, Aruba, August 2005

The authors also discuss how the memory size needed by a new job can be assessed. This is actually very similar to the technique of VMS that uses the 'Balance Sets' method. However, the authors of this paper take the same 'Balance Sets' concept for distributed systems.

In [9] the author suggests to tackle the thrashing effect by adjusting the memory needs of a process to the current available memory. This solution is quite different from the others, because it modifies the processes instead of modifying the operating system.

There are also some hardware solutions for thrashing which are implemented in the cache [10, 11]. Usually LRU is the basic scheme behind both hardware and software victim selection algorithms. However, the LRU algorithm is manipulated differently by hardware and software solutions. Obviously, hardware solutions must be much simpler for implementation. On the other hand, hardware solutions can use data that the operating system does not have, e.g. the cache can distinguish between an instructions block and a data block and this parameter might be useful for victim selection algorithms.

This paper suggests a technique that modifies the traditional process-scheduling procedure. This modification helps the operating system to swap in and out fewer pages; thus minimizing the slowdown stemming from thrashing. The solution suggested in this paper is not restricted to a specific operating system; hence, any multitasking paging system can implement it. The figures and the results given in this paper were achieved using the Linux operating system [12]. However, as we have noted, Linux is just a platform to show the feasibility of the concept.

The rest of the paper is organized as follows. Section 2 describes the Linux scheduling algorithms. Section 3 introduces the Bin Packing problem. Section 4 presents the reduced paging algorithm. Finally, Section 5 gives the results and evaluates them.

2. LINUX THRASHING HANDLING

The old UNIX scheduler is priority based [13]. The Linux process-scheduling algorithm is based on the old UNIX scheduler. More details about the Linux scheduler can be found for example in [14, 15] and in many other books and websites.

The virtual memory capabilities along with the paging mechanism give Linux the ability to handle many processes, even when the real memory needs are larger than those of the available physical memory. However, the virtual memory mechanism cannot handle every situation. If the memory pages demand is too high over a short period, the swapping mechanism cannot satisfy the memory needs reasonably. Pages are frequently swapped in and out and a little progress is made.

Linux only kills processes when thrashing occurs and the system is out of swap space. In some sense there is nothing else that the kernel could do in this case, since memory is needed but there is no more physical or swap memory to allocate [16, 17]. When such a case occurs, Linux kernel kills the most memory-consuming processes. This feature is very drastic; hence its implications might be severe. For example, if a server runs several applications with mutual dependencies, killing one of the applications may yield unexpected results.

The recent Linux 2.6 version has adopted the token-ordered LRU policy [18]. The basic concept of this policy is eliminating page swapping at some cases called by the authors 'false LRU pages'. Occasionally, a page of a sleeping process is swapped out, although it would have not been swapped out if the process were not sleeping. The idea of the token-ordered LRU policy is setting one or multiple tokens in the system. The token is taken by one process when a page fault occurs. The system eliminates the false LRU pages for the process holding the token and allows the process a quick establishing of its working set. By giving this privilege to a process, the total number of false LRU pages is reduced and an order is put in the pool of the competing pages. However, this policy can be helpful only if the memory needs slightly exceed those of the physical memory. A large memory excess by many processes will be treated by this method as FIFO, while the other processes still vie for memory allocations and thrash; hence, the authors of [18] suggest to keep the traditional killing approach of Linux for severe cases. We suggest another approach that can also handle these severe cases.

In addition, the process selection algorithm of Linux can mistakenly select a process that executes an endless loop. Such a selection will worsen the thrashing. Even selecting a very long process that is executed for some hours will be harmful. The selection algorithm can just guess which the shortest process is, but this guess may be wrong.

3. BIN PACKING

The suggested technique needs a set of all the processes that are currently in the virtual memory. This set is split into several groups, such that the total memory size of each group is as close as possible to the size of the available real memory.

How can we build these groups of processes? We have a set of processes P_i, each with a memory allocation. Let M_i denote the maximal working set size that might be needed by process P_i. We need an algorithm which splits these M_is into as few groups as possible, with the sum of the M_is in each group not exceeding the size of the real memory. Practically, the kernel and some other daemons occupy part of the memory, so the sum should not exceed a smaller memory size. This splitting problem is well known and is called the Bin Packing problem [19].

The Bin Packing problem is defined as a set of numbers $X_1, X_2, \ldots, X_n$, with $X_i \in [0, 1]$ for each i. The problem is finding the smallest natural number m for which

- $X_1, X_2, \ldots, X_n$ can be partitioned into m sets.
- The sum of the members of each set is not higher than 1.

The Bin Packing problem is NP-hard [20]. However, some polynomial time approximations have been introduced over the years, such as those in [21, 22, 23, 24]. The approximation algorithms use no more than $(1 + E)$*OPT(I) number of bins, where OPT(I) is the number of bins in the optimal solution for case I. If E is smaller, the result will be closer to the optimal solution, but unfortunately good approximations are usually time-consuming [25]. We would like to choose one of the approximation algorithms which is not time-consuming, yet tries minimizing $(1 + E)$*OPT(I).

A simple idea of an approximation algorithm for the Bin Packing problem is the greedy approach [26], also known as the First-Fit approach. This algorithm is defined as follows:

- Sort the vector $X_1, X_2, \ldots, X_n$ by the allocated memory size.
- Open a new bin and put the highest number in it.
- While there are more numbers
  - if adding the current number to one of the existing bins exceeds the size of the bin
    - open a new bin and put the current number in it.
  - Else
    - put the current number in the current bin.

In our tests, we used a version of this approximation algorithm with a slight modification. We usually achieved the minimal number of bins and the cost of execution time was usually low. The version we used is described below.

4. BIN PACKING-BASED PAGING

It is well known that increasing the level of multitasking in any operating system may sometimes cause thrashing. In order to avoid thrashing, we would like to suggest a new approach: all the processes will be split into several groups such that the sum of physical memory demands within each group will not be higher than the amount of physical memory available on the machine. In [27] the authors give some ideas to use a Bin Packing approximation (First-Fit) to improve the Backfilling scheduling of a specific operating system (Tera). We would like to use the Bin Packing algorithms to improve the Linux scheduling using more approximations.

4.1. The medium-term scheduler

A new scheduler procedure will be added to the Linux operating system. The new scheduler will operate in the manner of the medium-term scheduler, which was part of some operating systems [28]. The medium-term scheduler will load the groups into the Ready queue of the Linux scheduler in a Round-Robin manner. The traditional Linux scheduler will do the scheduling within the current group in the same way the scheduling is originally done on Linux machines. The time slice of each group in the medium-term scheduler will be significantly higher than the average time allocated to the processes by the original Linux scheduler. The processes in the real memory will not be able to cause thrashing during the execution of the group, because their total size is not higher than the size of the available physical memory, i.e. the size of each bin. Only at the beginning of each group execution there will be an intensive swapping, because the new group's pages are swapped into the memory. This approach can improve the system ability to support memory-consuming processes in a more tolerant way than killing them.

There are some methods to calculate the working set size needed by each process. One of these methods can be found in [29]. In this paper, the authors suggest a way that adds 7–10% overhead. Obviously, such an overhead is time-consuming and not suitable for the concept of the Bin Packing approach. The scheduler needs to know the working set size on every context switch and calculating this working set often is costly. We use another simple approach. The resident size of each process was taken from /proc/PROCESS_NO/status file. This size is the process' last pages total size. This size is not accurate and if the system is not busy, the resident size may include large portions of stale pages that are not currently essential. However, when the system is not busy, there will be no thrashing and this overestimation will make not be harmful.

In our implementation, the group time slice was half-a-second or one second, whereas the Linux scheduler gives time slices of some dozen milliseconds. When Linux thrashes, any context switch causes many page faults, whereas with the medium-term scheduler, intensive swapping will occur only when switching between groups. This allows the operating system in our implementation to swap a significant number of pages in only a few percent of cases, in contrast to conventional Linux during thrashing conditions.

The processes which are not in the current group should be kept on a different queue, so that Linux scheduler will not be able to see them. In order to implement this feature, we added a new record to Linux kernel code. This record has the same structure as the 'active' and 'expired' records described in [14] and it holds the hidden processes.

When the last group finishes its execution, the medium-term scheduler is invoked, and rebuilds the process groups, taking into account any changes to the old processes (e.g. exited or stopped) and adding any new processes to the groups.

Sometimes the current group finishes executing all the processes within the time slice awarded to it by the medium-term scheduler. Even if there are still some processes in the group, these processes might be sleeping. If not all the processes in the group are ready to be executed, the Linux scheduler has

been modified to invoke the medium-term scheduler, which promptly switches to the next waiting group.

The medium-term scheduler takes the sum of the memory sizes that are currently needed by the processes and divides this sum by the available physical memory size. The quotient is taken to be the number of bins. After that, the medium-term scheduler scatters the processes between these bins. The medium-term scheduler uses the greedy algorithm until the medium-term scheduler is unable to fit another process into the bins. Next, the medium-term scheduler tries to find room for all the remaining processes in the existing bins. If it fails to find room in one of the existing bins, it exceeds the size of the smallest bin by adding the unfitting process to it. The original Bin Packing problem does not allow such an excess, but in this case it might be preferable to have a few page faults within a group than adding an additional bin.

One of our assumptions for a working solution is that there exist a considerable number of processes for a good bin packing, and some small memory demand processes are even better. However, if one process demand is larger than the available memory size, the solution will not be effective and the process will thrash within itself. In fact, most of the thrashing cases are not caused by one process. However, if such a case does occur, none of the solutions that has been presented in Section 1 can be useful. In such a case, only the original Linux solution that kills such a process will be beneficial, but it can be harmful if the process is essential.

4.2. Swapping management

When the time slice of a group ends, a context switch of groups will be performed. This context switch will probably cause many page faults: the kernel uses its swap management to make room for the processes of the new group and this procedure might be long and fatiguing. The previous group of processes has most probably used up most of the available physical memory, and when the swap thread executes the LRU function to find the best pages to swap out to the disk, it will probably find pages of the old group. This procedure is wasteful because the paging function is performed separately for every new required page. Linux kernel does not know at the context switch time that the recently used pages of the previous group will not be needed for a long time, and can be swapped out.

In order to overcome this Linux kernel management, we modified Linux kernel as follows: when the medium-term scheduler is invoked, it calls the Linux swap management functions to swap out all of the pages that belong to the processes of the previous group. This gives Linux a significant amount of empty frames for the new group. This swapping management approach is much quicker than incrementally loading the pages of the new process group, and for each page fault searching for the oldest page in the physical memory to swap out. When a round of the medium-term scheduler is completed, the medium-term scheduler will rebuild the process groups and some processes may migrate from one group to another; hence the medium-term scheduler does not call the Linux swap management at this point, because it might swap out pages that may be needed again for the next group.

4.3. Shared memory

Often, two or more processes share some memory. Shared memory widely exists in most of the operating systems and Linux has some tools to handle it too.

When using the medium-term scheduler, it will be inefficient to put two processes that share a large piece of memory into two different groups of processes. For example, consider the following scenario:

Suppose processes A and B share a piece of memory, process A is of group #1 and process B is of group #3. After group #1 completes its execution, the pages of group #1 are swapped out and the pages of group #2 are loaded and placed in the physical memory instead, as we explained in the previous section. The same swapping happens when the operating system replaces group #2 by group #3. The pages of process B are loaded by demand, so the same pages, albeit not all of them, are loaded and swapped twice, once for process A and once again for process B.

This situation is illogical and inefficient; hence, the medium-term scheduler must put processes with a large shared memory in the same group.

In order to tackle this situation we must know which processes share pages, and which pages they share. Each process that uses a page increments the ‘count’ field of the page [30], so reading this field can easily let the Linux know which pages are shared. However, the medium-term scheduler must still know which processes use the page. A naive approach would be searching the page table of all the processes, in order to extract the addresses of the given pages. Obviously, this will be very time-consuming and will probably impair the medium-term scheduler efficiency.

It frequently happens that applications which share a piece of memory will have an almost equal size of shared memory. If the shared memory stems from a ‘fork’ system call, the child process will be created from its parent; hence the size of the shared memory in the parent process and the child process will be almost equal, unless the child or the parents allocate a large piece of new shared memory. The same will be correct, if processes share an IPC or a common text segment. If there is no other shared memory obtained by just one of the processes, the shared memory size will be almost equal.

Figure 1 shows the shared memory size in some common cases. The data was obtained from a running Linux machine that serves the computer science department in Bar-Ilan University. Many processes do not have shared memory and they have been omitted from the figure. However, when processes do share memory, it can be seen that they usually have the same size of shared memory.

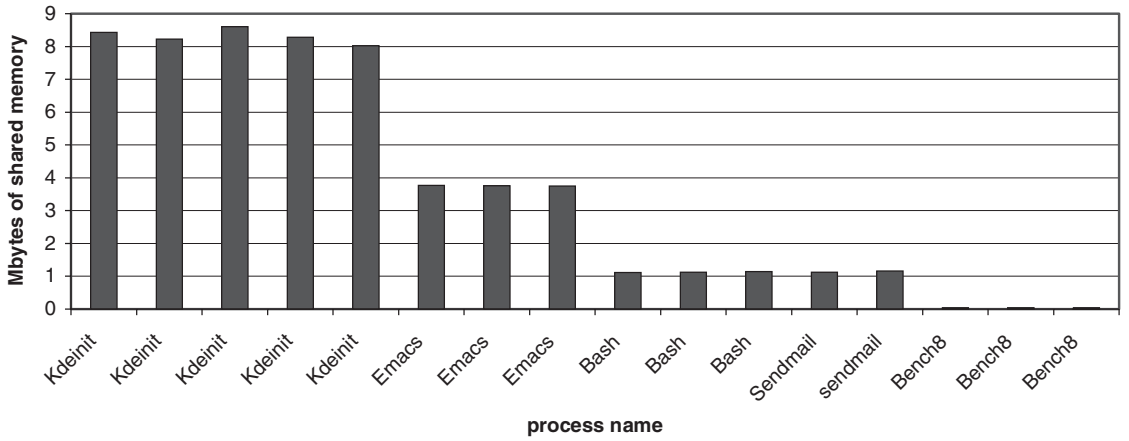

FIGURE 1. Shared memory size of common processes.

Linux calculates the shared memory size of each process. Based on the shared size characteristic, we would like to suggest a simple solution for the issue of large shared memory. When the medium-term scheduler recalculates the bins of the processes, it will first sort the vector of the processes by the shared memory size. Then, using the greedy approximation, the processes with almost equal shared memory size will usually be in the same bin or in adjacent bins; hence no swapping will be needed when replacing the pages of two processes with shared memory.

We have chosen to use insertion sort for this. Since we use the old sorted list of processes, insertion sort is executed in the shortest time [31].

4.4. Group time slice

Sometimes we can be lucky and the sizes of the total memory needs in all the different groups are almost equal. This is the best situation, because a fixed time slice that will be given to the groups is usually quite fair. However, when the sizes of the total memory allocations are significantly different, some processes might get an implicit high priority. When the medium-term scheduler uses the greedy approximation, such a situation usually occurs when the last processes are assigned to a bin. The last bin is sometimes almost empty; hence the processes in this bin gain precedence, because in the time slice of this bin, there are less processes vying for CPU cycles. It should be noted that when the size of the last bin is not small, this solution will function efficiently.

One possible solution is breaking up small groups and scattering processes that belong to the small groups in other groups. This solution can be good if the size of the small group is not big and when there is just one small group. If the size of the small group is big, scattering it might cause thrashing in the other groups.

A better solution can be a dynamic group time slices, instead of a constant time slice. For example if the size vector is [1,1,0.5] and the default group time slice is one second, the medium-term scheduler should assign each of the first two groups one second, whereas the last group will get only 0.5 s. (The vector represents the group's memory size as the total memory allocations divided by the total memory available for user application). This solution gave us the best results; therefore it has been implemented.

4.5. Interactive processes

The interactive processes should be dealt with differently. If we treat them the same way as the non-interactive processes, they will not be able to be executed as long as their group is not current. Interactive processes need fast response time and a few seconds delay can be a major drawback.

To remedy this drawback, the scheduler allows an interactive process which can be identified by directly quantifying the I/O between an application and the user (keyboard, mouse and screen activity) [32] to run in each of the process groups. So, actually the process will belong to all the groups, but with a smaller time slice in each group:

$$\mathrm{p} \rightarrow \mathrm{time\_slice} = \frac{\mathrm{time\_slice(p)}}{\mathrm{num\_of\_groups}}.$$

This feature can assure us a short response time for interactive processes while keeping fairness towards other processes. The resident pages of interactive processes will be marked as low priority swappable, so the kernel will not swap out interactive processes when a group context switch is done. However, the scheduler has to calculate the memory needs of interactive processes in every group.

When a new process is admitted, it will be handled as an interactive process. The operating system cannot know whether the new process is interactive and if the execution of this process is delayed, it will be irritating for interactive processes. After one round of the bins, the scheduler can assess the nature of the process and treat it accordingly.

4.6. Real-time processes

The handling of real-time processes is somewhat similar to interactive processes. Real-time processes must get the CPU as fast as possible. The management of these processes will be the same as interactive processes, but with a slight difference. Real-time processes will belong to all the groups, as the interactive processes do, but they will not have a shrunken time slice.

The kernel will not swap out real-time processes, because they belong to all the groups. In addition, real-time processes will have the same privilege Linux traditionally gives them. It should be noted that the scheduler has to calculate their memory needs in every group as the scheduler did for the interactive processes. This handling is identical for FIFO real-time processes and for RR real-time processes. This treatment has also been applied to the 'init' process and the 'Idle and Swapper' process of Linux, which cannot be suspended.

4.7. Priority

Another important issue of the Bin Packing scheduling discussion is the priority management. Hypothetically, it might happen that the highest priority processes belong to one group, whereas the lowest priority processes belong to another. Then, when Linux switches between the processes within the groups, the priority is not taken into account.

One solution can be finding out how many bins there should be, by calculating the total size of the memory needs and dividing by the size of the available physical memory (the size of the bin), just as the medium-term scheduler always does, then sorting the process list by priority, and finally taking the processes from the sorted list and filling the bins in a Round-Robin manner. This solution cannot be implemented together with the shared pages solution, because the shared pages solution requires sorting by the number of the shared pages, rather than by the priority.

Another solution is assigning a different time slice to each group, according to the average priority of the processes inside the group. For each group the average priority is calculated. A group having a high average priority will be awarded a longer time slice. This solution was chosen based on the results that are shown in Section 5.4.

5. EVALUATION AND RESULTS

5.1. Testbed

We tested the performance of the kernel with the new scheduling approach using five different benchmarks to get the widest view we could

(i) SPEC—cpu2000 [33]. The SPEC manual explicitly notes that attempting to run the suite with less than 256 MB of memory will cause a measuring of the paging system speed instead of the CPU speed. This suits us well, because our aim is precisely to measure the paging system speed; hence, we used a machine with just 128 MB of RAM. Using machine with a larger RAM would have forced us not to use SPEC.

(ii) A synthetic benchmark that forks processes which demand a constant number of pages—8 MB. The processes use the memory in a random access; therefore they cause thrashing. This benchmark was tested within the range of 16–136 MB. The parent process forks processes whose total size is the required one and collects the information from the children. Let us denote this test by SYN8.

(iii) Matlab formal benchmark. This benchmark executes six different Matlab tasks described in [34].

(iv) Another synthetic benchmark using massive shared memory allocations. The test has two processes that share 16 MB and has 2 more MB for each one of the processes. The processes copy parts of their private memory into the shared memory and parts of the shared memory into their private memory in a random access. The benchmark consists of a number of such tests according to the desired size. Let us denote this test as SYNSHARED.

(v) For interactive and real-time processes, we used the Xine MPEG viewer. It was used to show a short video clip in a loop.

The benchmarks were executed on a Pentium 2.4 GHz with 128 MB RAM and a cache of 1 MB running Linux kernel 2.6.9 with Fedora core 2 distribution. The size of the page was 4 KB. It should be noted that even though the platform machine had 128 MB of physical memory, we should take into the bin size considerations that a certain portion of this memory is occupied by the daemons of Linux/RedHat and the X-windows, plus the kernel itself along with its threads. After an evaluation of the extra size, we used bins of 96 MB.

5.2. Execution time

Figure 2a and b show the performance of the synthetic benchmark SYN8. Figure 2a shows the number of swaps that were performed in both the schedulers as a function of the total size of the processes, whereas Figure 2b shows the execution time of SYN8 as a function of the same processes' total size. In these figures, the medium-term scheduler time slice was one second.

It can be seen that when the size of the processes is too large, Linux starts swapping in and out many more pages. From roughly 64 MB Linux swaps more pages, but there is no noteworthy influence on the I/O time, because Linux lets other processes run while the I/O is performed. Roughly, from 128 MB the I/O buffer is incapable of responding to all the paging requests, and the thrashing becomes acute.

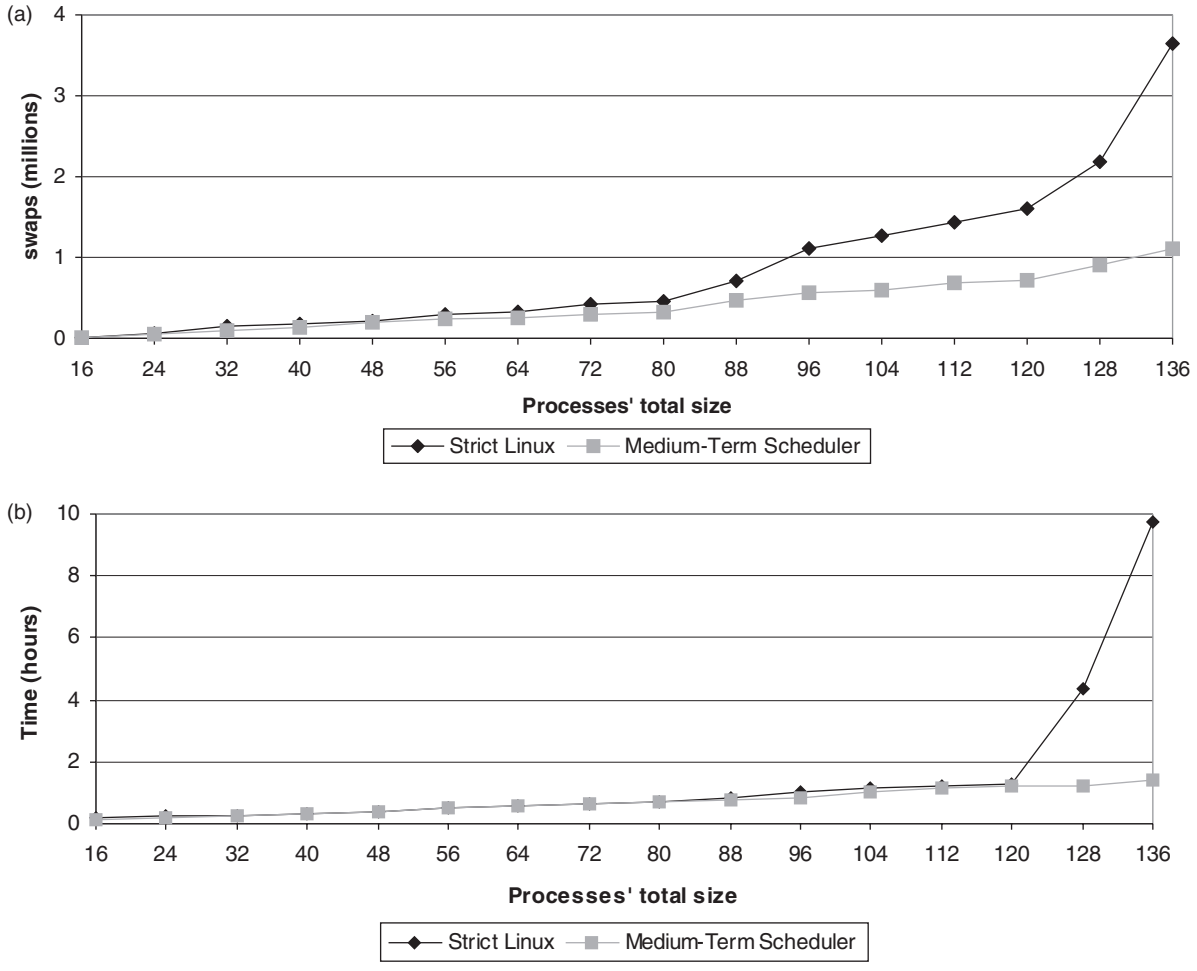

FIGURE 2. (a) SYN8's number of swaps. (b) SYN8's execution time.

The medium-term scheduler dramatically reduces the number of page faults; thus, fewer swaps are performed and the execution time remains reasonable. Processes that require 144 MB or more were sustainable for the medium-term scheduler, but not for the Linux scheduler.

We also employed Matlab formal benchmark. Matlab benchmark is a very memory-consuming process. It takes ∼290 MB with Matlab 7.0.0.19901 (R14) running on our Linux 2.6.9 machine, but when memory pressure becomes high, Matlab will be able to continue working when just 28 MB are resident in the physical memory, whereas 14 MB of them are shared memory with other possible Matlab processes. When we executed several Matlab processes in parallel, the results were very similar to the synthetic benchmark. However, a significantly larger portion of the swap area was necessary, because just 14 MB out of each Matlab process was physically in the internal memory and the other memory allocations (except of the shared allocation) of the Matlab processes were in the swap area. We preferred not to reshow the results that are almost the same as those in Figure 2a and b and instead preferred to show in the following figures different benchmark results.

Figure 3a and b show the performance of the medium-term scheduler versus the Linux kernel using the tests of SPEC cpu2000 benchmarks. The prefix 3 (or 2) before the test name indicates that we iterated the test 3 (or 2) times. Sometimes we divided the numbers by some constants in order to fit the data to the scale of the diagram. These constants are denoted as Test/Constant. When we used more than one test, we added a '+' sign between the names of the tests.

When each group contains just a few memory-consuming processes, the idle task might be invoked too often, even though there are other processes in other groups that can be executed. This can reduce the time saved by eliminating the thrashing effect. When a test has large memory allocations and is executed in a different group, the results will not be as good as when executing several smaller SPEC tests concurrently in one group. A higher idle time will emerge when the content of each group is just one process; thus the results of

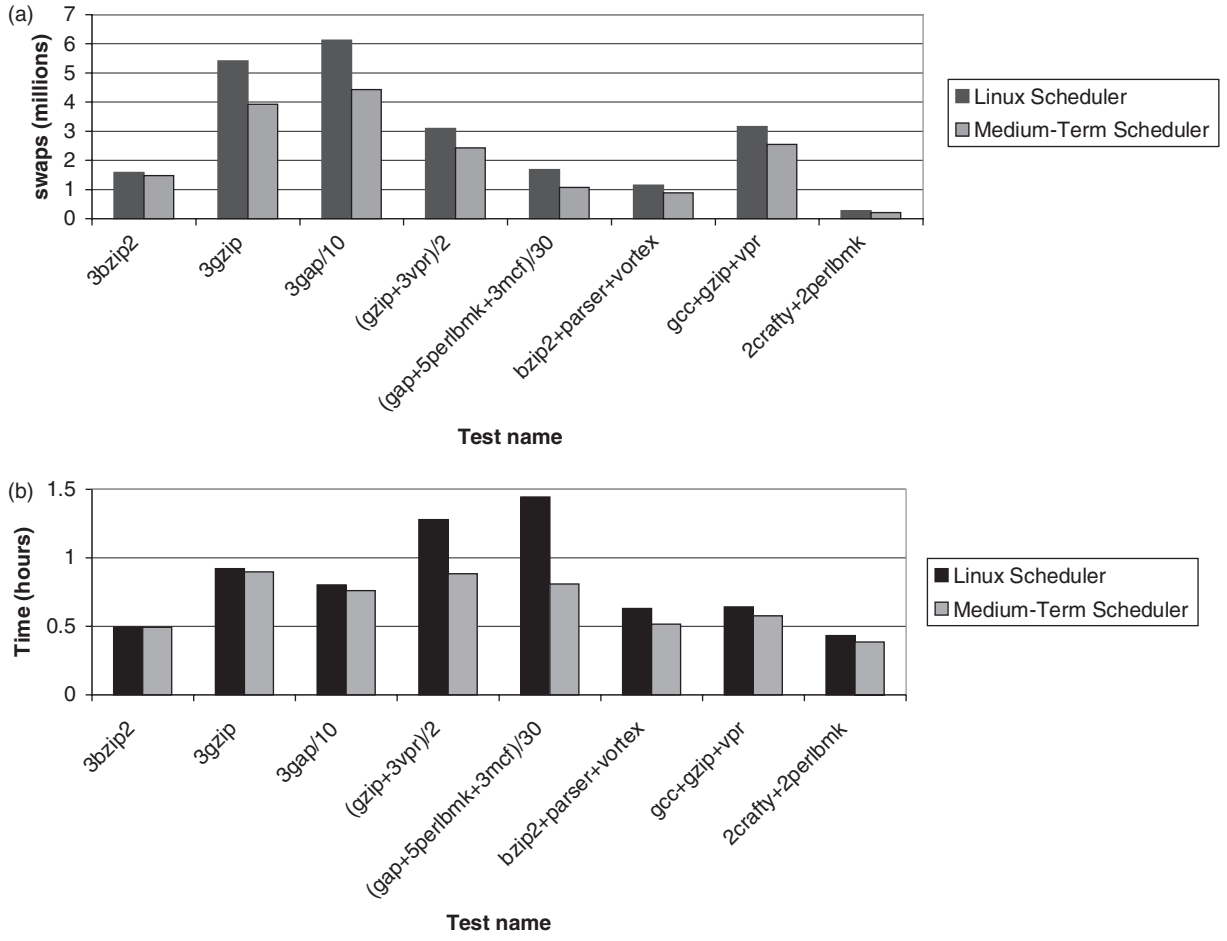

FIGURE 3. (a) SPEC's number of swaps. (b) SPEC's execution time.

Figure 3a and b are not as good as the results of Figures 2a and b. However, the elimination of the thrashing saved more time than was wasted idling, and the medium-term scheduler still outperforms the traditional Linux scheduler.

Figure 4a and b show the effect of the medium-term scheduler time slice on the process' execution time. The tests were conducted using SPEC. It can be seen that when the time slice exceeds a certain limit, the execution time might suffer. This damage is caused by the higher average idle time. When the number of processes per group is too small, it may happen that none of the processes in the current group is on the Ready queue. Such a case may happen due to many I/O operations. Clearly, this might turn out with a lower group time slice as well, but it will not happen as often as with a higher time slice, because at the beginning of the time slice all the processes are usually ready to run and not waiting for an I/O.

When the time slice is higher, the cycle will be longer. An extremely high time slice will actually make the medium-term scheduler behave like an FIFO scheduler. On the other hand, the page faults rate is lower for the one second scheduler, because of the longer time slice. Pages are usually swapped out when the group context is switched, so if all the pages are replaced on context switch, the half-a-second scheduler should have double the number of page faults compared with that of the one second scheduler. However, sometimes the bins are not full, and some shared memory can be present, so the ratio between the number of page faults is actually less than 2.

Figure 5 shows the same time slices but with more processes. This test was conducted using the synthetic benchmark SYN8. It can be clearly seen that the effect of increasing time slice damages the execution time when processes for more than one bin are present.

5.3. The Bin Packing approximations

There are more than a few approximations for the Bin Packing optimal solution. Some of them have been mentioned in Section 3. Figure 6a and b compare two of these approximations. The First-Fit approximation (also known as the greedy approximation) is described in Section 4.1. The Best-Fit Approximation finds for each process the most unfilled bin and put the process in it.

When there is almost no shared memory, the performance of both methods will be almost the same. However, when a

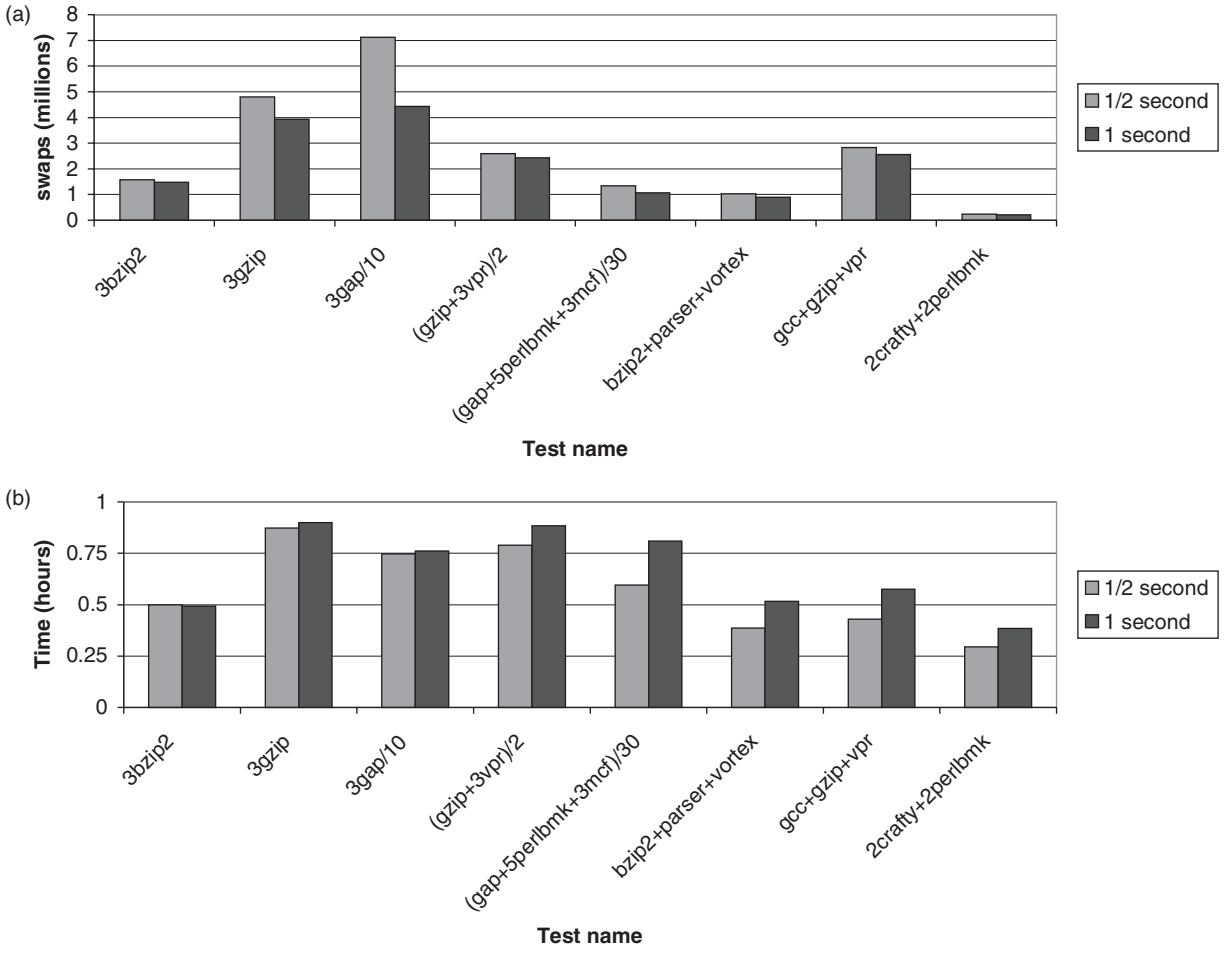

FIGURE 4. (a) Time slice's effect on number of swaps. (b) Time slice's effect on execution time.

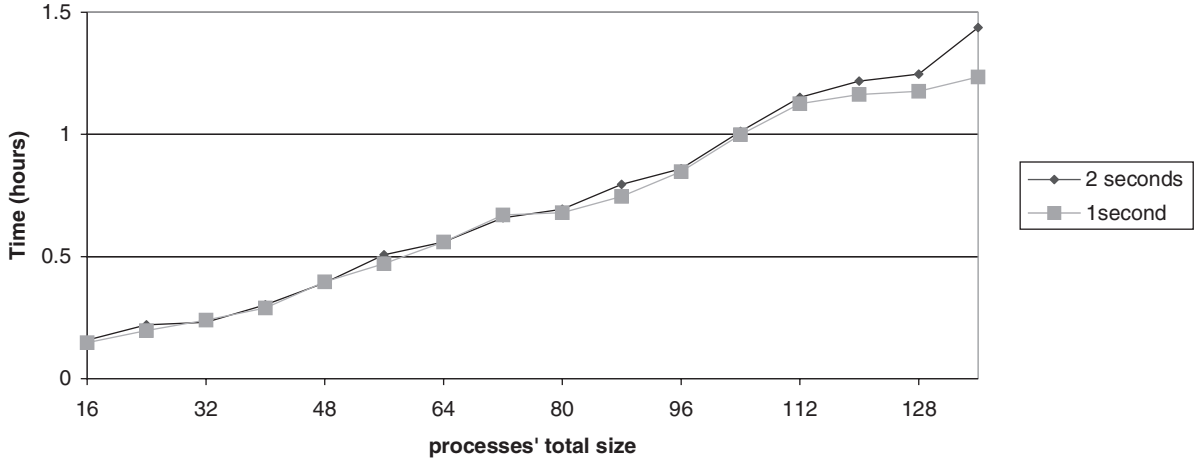

FIGURE 5. Execution time as a function of processes' total size.

significant amount of shared memory is allocated, the First-Fit approach outperforms the Best-Fit approach. The benchmark that was used in Figure 6a and b is SYNSHARED. Figure 6a compares the number of swaps using each of the methods. First-Fit sorts the processes according to their shared size; hence usually processes that share a portion of memory will be in the same group. As was explained in Section 4.3 processes that share memory typically have the same number

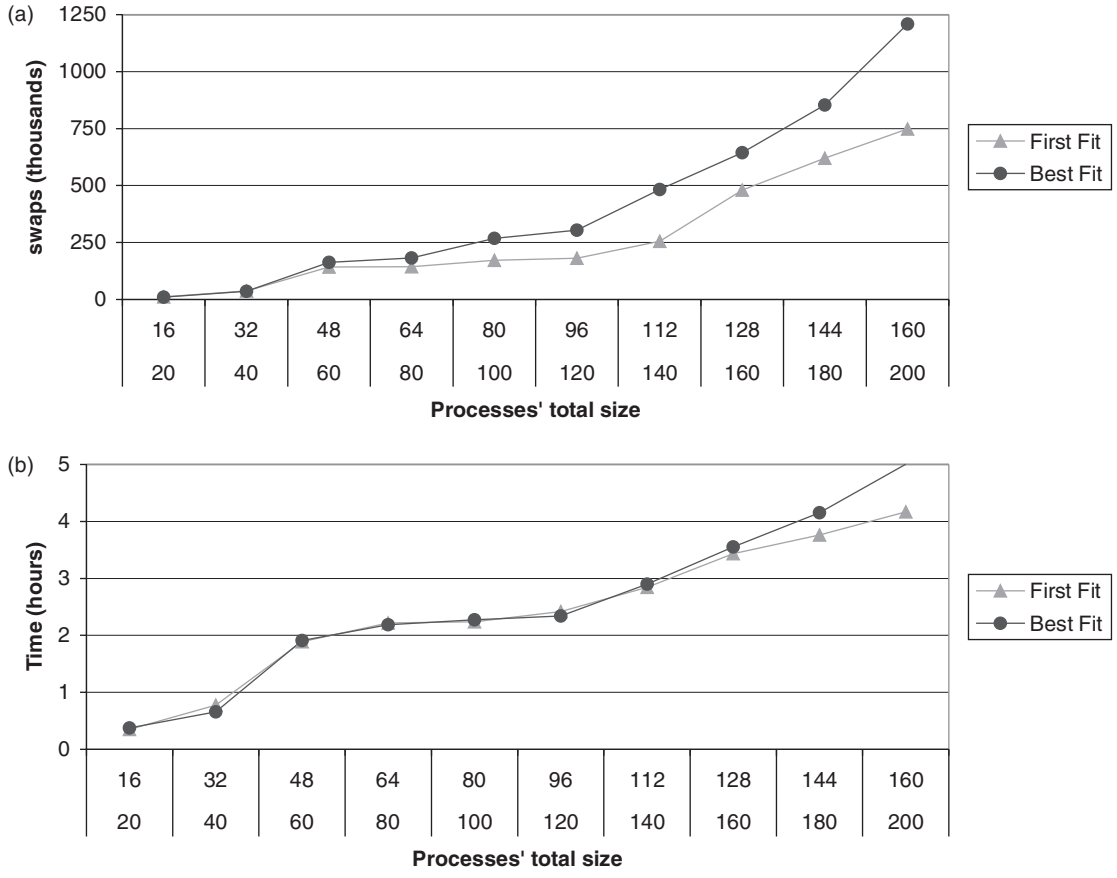

FIGURE 6. (a) Number of swaps using Best-Fit or First-Fit. (b) Execution time using Best-Fit or First-Fit.

of shared pages. As a result, they will be in adjacent positions in the sorted list and probably will be put in the same group. Therefore, less page faults will occur. Best-Fit cannot guarantee this quality; hence, the performance will not be as good as that of First-Fit. The higher number of page faults causes a longer execution time as can be seen in Figure 6b.

5.4. Evaluation of the priority implementations

The priority can be implemented by another approximation which first determines how many bins there should be. Next, it sorts the processes by their priority and finally, it fills the bins in a Round-Robin manner. This method can scatter the higher priority processes (and the lower priority processes) among the bins, more or less equally. However, the shared memory handling requires a sorting by the shared size; hence, if there are many processes with shared memory allocations, this approach can lengthen the execution time.

Another approach implements the priority by dynamically changing the time slice according to the average priority of the processes in the group. This approach sorts the processes by their shared memory sizes and builds bins using a First-Fit version that has been introduced above. Actually, this approach performs the same procedure of building the bins, but each group gets a dynamically different time slice, according to the average priority of the group. The medium-term scheduler calculates the global average priority of all the processes currently run and the average priority of the processes in each group. Next, it calculates the difference between the average of each group and the global average. Let us denote this vector of differences as D and the global average priorities of all the processes as P. Then, the medium-term scheduler gives each group $(D[i] + P)/P^*\text{TS}$, where TS is the default group time slice and i is the index the group.

Figure 7 shows the differences between the approaches. We used the SPEC benchmark. We took in each test one process and we awarded it the highest priority: -20. We always took another process and demoted its priority to the lowest one: 19. The tests are written on the x-axis. The promoted test is written below. We did not change any other process' priority. The default group slice time was one second.

The differences between the two strategies can be clearly shown when using the SYNSHARED benchmark. Because of the massive use of shared memory, the sorting by shared memory strategy will dramatically outperform the sorting by priority strategy. The results of the SYNSHARED are shown in Figure 8a and b. Figure 8a shows the influence of the strategies on the number of swaps. This is quite a dramatic

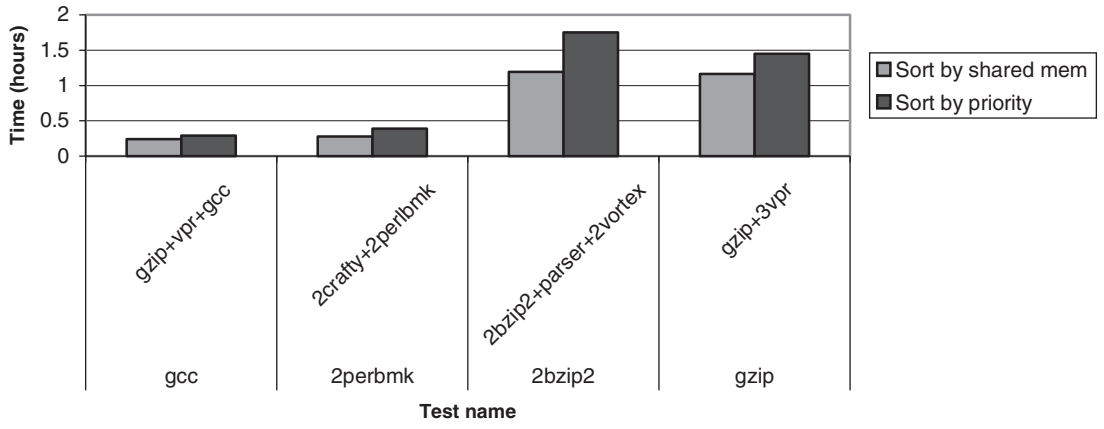

FIGURE 7. Different sorting strategies of the medium-term scheduler.

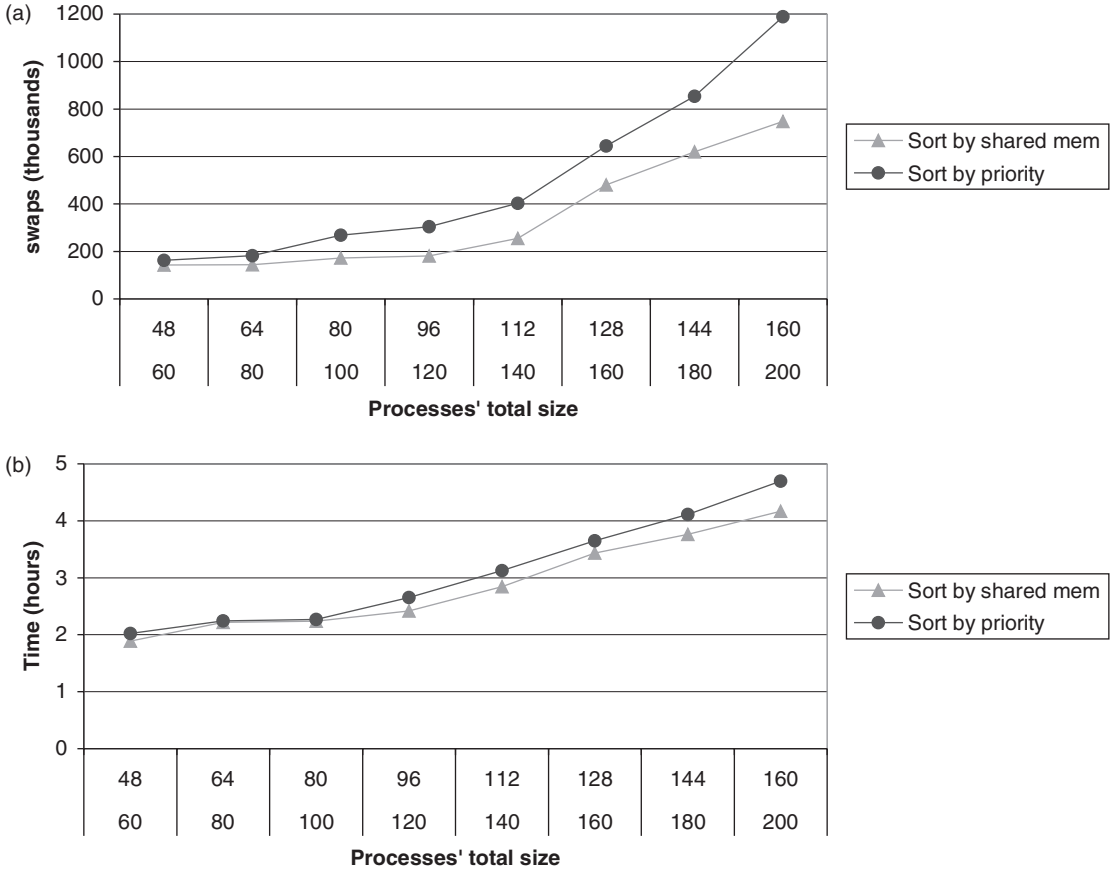

FIGURE 8. (a) SYNSHARED's number of swaps with different sorting methods. (b) SYNSHARED's execution time with different sorting methods.

difference. The difference of the execution time is notable as well.

5.5. Interactive and real-time processes

The interactive and real-time processes were checked using the Xine movie player. It is a well-known MPEG player on Linux machines. We configured Xine to play a short video clip in a loop. The memory needs of Xine are much lower than those of the physical RAM we had in our machine. In order to check that Xine will continue to respond even when the memory is overloaded, we deliberately overfilled the memory by executing many copies of SYN8. The results of this test can be found in Figure 9. When the movie player process is not

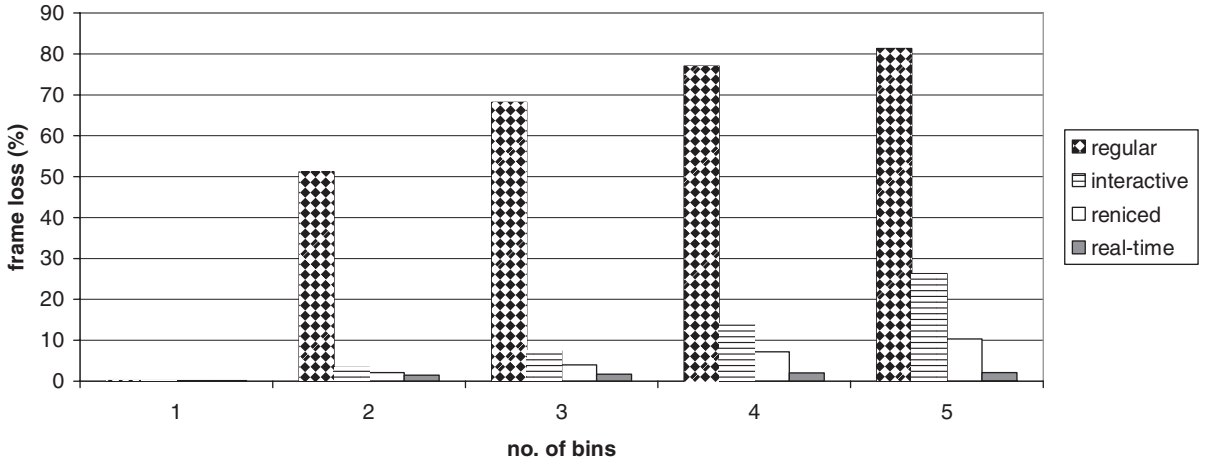

FIGURE 9. Frame loss as function of the number of bins.

handled as an interactive process, many frames are lost. When Xine's bin is not active, no CPU time is given and no frames can be displayed. Even when a CPU time is given, if the slice is reduced because of the overall load, sometimes the given slice is not enough and just when the process is handled as a real-time process, a good result can be achieved. We also reniced Xine by -20. This yielded interesting results. The results were better than those of the interactive mode, because interactive processes' time slice is reduced when there are too many bins, whereas the reductions of the time slice of bins include a high priority process which is smaller. On the other hand, a high priority process does not have the privileges Linux gives to real-time processes, so the results are worse than those of the real-time mode.

6. CONCLUSIONS AND FUTURE WORK

The results of the experiments are promising. Given a high memory load used by some processes, the medium-term scheduler can drastically reduce the thrashing overhead. In addition, no decline in the performance is observed when the load is low and no swapping is needed. The medium-term scheduler has been written as a patch to the kernel and can be easily installed on any Linux machine. Such an installation can help the machine to handle the massive paging in a more tolerant way than to kill processes. Moreover, the responsiveness keeps being reasonable for heavier load. The medium-term scheduler does not require special resources or extensive needs; hence, it can be easily adapted by many Linux machines. Furthermore, there is no obstruction to implement the medium-term scheduler on a cluster; hence, heavy load projects like the Human Genome project can benefit from such a kernel.

In the future, we would like to check the performance of other approximations for the Bin Packing problem and even to adaptively change the approximation according to the current conditions in the system. In addition, we would like to dynamically change the group time slice of the medium-term scheduler. This feature can improve the performance when there are too few processes and the idle process is invoked too often.

REFERENCES

[1] Zahorjan, J., Lazowsk, E. and Eager, D. (1991) The effect of scheduling discipline on spin overhead in shared memory multiprocessors. *IEEE Trans. Parall. Distr. Syst.*, **2**(2), pp.180–198.

[2] Denning, P. (1970) Virtual memory. *ACM Comput. Surv.*, **2**(3), 153–189.

[3] Belady, L. A. (1966) A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, **5**(2), 78–101.

[4] Abrossimov, V., Rozier, M. and Shapiro, M. (1989) Virtual memory management for operating system kernels. In *Proc. 12th ACM Symp. Operating Systems Principles*, Litchfield Park, AZ, December 3–6, pp. 123–126. ACM SIGOPS, New York.

[5] Galvin, P. B. and Silberschatz, A. (1998) *Operating System Concepts* (6th edn). Addison Wesley Longman, Harlow, MA.

[6] Jiang, S. and Zhang, X. (2001) Adaptive page replacement to protect thrashing in Linux. *Proc. 5th USENIX Annual Linux Showcase and Conf.,* (ALS'01), Oakland, CA, November 5–10, pp. 143–151. USENIX, Berkeley.

[7] Jiang, S. and Zhang, X. (2002) TPF: a system thrashing protection facility. *Softw. Pract. Exp.*, **32**(3), 295–318.

[8] Batat, A. and Feitelson, D. G. (2000) Gang scheduling with memory considerations. In *Proc. 14th Int. Parallel and Distributed Processing Symp. (IPDPS'2000)*, Cancun, Mexico, May 1–5, pp. 109–114. IEEE, Los Alamitos.

[9] Nikolopoulos, D. S. (2003) Malleable memory mapping: user-level control of memory bounds for effective program adaptation. *Proc. 17th Int. Parallel and Distributed Processing Symp. (IPDPS'2003)*, Nice, France, April 22–26, volume published in CD-ROM. IEEE, Los Alamitos.

[10] Gonzalez, A., Valero, M., Topham, N. and Parcerisa, J. M. (1997) Eliminating cache conflict misses through XOR-based placement functions. In *Proc. Int. Conf. Supercomputing*, Vienna, Austria, July 7–11, pp. 76–83. ACM, New York.

[11] Chu, Y. and Ito, M. R. (2000) The 2-way thrashing-avoidance cache (TAC): an efficient instruction cache scheme for object-oriented languages. In *Proc. 17th IEEE Int. Conf. Computer Design (ICCD2000)*, Austin, TX, September 17–20, pp. 93–98. IEEE, Los Alamitos.

[12] Card, R., Dumas, E. and Mevel, F. (1998) *The Linux Kernel Book*. John Wiley & Sons, New York.

[13] Vahalia, U. (1996) *UNIX Internals: The New Frontiers*. Prentice Hall, New Jersey, pp. 112–148.

[14] Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R. and Verworner, D. (1998) *Linux Kernel Internals*. (2nd edn). Addison Wesley, Longman, Harlow, MA.

[15] Komarinski, M. F. and Collett, C. (1998) *Linux System Administration Handbook*. Prentice Hall, New Jersey.

[16] Gorman, M. (2004) *Understanding The Linux Virtual Memory Management*. Chapter 13. Bruce Peren's Open Book Series, New Jersey, pp. 117–122.

[17] Marti, D. (2002) System development jump start class. *Linux J.*, **7**.

[18] Jiang, S. and Zhang, X. (2005) Token-ordered LRU: an effective page replacement policy and implementation in Linux systems. *Perform. Evaluat.*, **60**(1–4), 5–29.

[19] Scholl, A., Klein, R. and Jurgens, C. (1997) BISON: a fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Comput. Oper. Res.*, **24**, 627–645.

[20] Karp, R. M. (1972) Reducibility among combinatorial problems. In Miller, R. E. and Thatcher, J. M. (eds), *Complexity of Computer Computations*. pp. 85–103. Plenum Press, New York.

[21] Fekete, S. P. and Schepers, J. (2001) New classes of fast lower bounds for bin packing problems. *Math. Program.*, **91**(1), 11–31.

[22] Fleszar, K. and Hindi, K. (2002) New heuristics for one-dimensional bin packing. *Comput. Oper. Res.*, **29**(7), 821–839.

[23] Gent, I. (1998) Heuristic solution of open bin packing problems. *J. Heuristics*, **3**, 299–304.

[24] Martello, S. and Toth, P. (1990) Lower bounds and reduction procedures for the bin packing problem. *Discrete Appl. Math.*, **28**, 59–70.

[25] Coffman, E. G. Jr, Garey, M. R. and Johnson, D. S. Approximation algorithms for bin packing: a survey. In Hochbaum, D. (ed.), *Approximation Algorithms for NP-Hard Problems,* pp. 46–93. PWS Publishing, Boston.

[26] Albers, S. and Mitzenmacher, M. (2000) Average-case analyses of first fit and random fit bin packing. *Random Struct. Algor.*, **16**, 240–259.

[27] Alverson, G., Kahan, S., Korry, R., McCann, C. and Smith, B. (1995) Scheduling on the Tera MTA. *Proc. 1st Workshop on Job Scheduling Strategies for Parallel Processing,* in conjunction with *IPPS '95* Fess Parker's Red Lion Resort, Santa Barbara, CA, April 25, pp. 19–44. Springer-Verlag, Berlin.

[28] Stallings, W. (1998) *Operating Systems Internals and Design Principles* (3rd edn), p. 383. Prentice-Hall, New-Jersey.

[29] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y. and Kumar, S. (2004) Dynamically tracking miss-ratio-curve for memory management. *Proc. Eleventh Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, Boston, MA, October 7–13, pp. 177–188.

[30] Bovet, D. and Cesati, M. (2003) *Undersatnding the Linux Kernel* (2nd edn), Chapter 7. O'Reilly Press, Sebastopol, CA, pp. 216–232.

[31] Manber, U. (1989) *Introduction to Algorithms—A Creative Approach,* pp. 130–131. Addison-Wesley, Harlow, MA.

[32] Etsion, Y., Tsafrir, D. and Feitelson, D. G. (2004) Desktop scheduling: how can we know what the user wants? *Proc. 14th ACM Int. Workshop on Network & Operating Systems Support for Digital Audio & Video (NOSSDAV'2004)*, Cork, Ireland, June 16–18, pp. 110–115. ACM, New York.

[33] SPEC (2000), *CPU-2000*. Standard Performance Evaluation Corporation, Warrenton, VA. Available at http://www.spec.org/.

[34] BENCH—MATLAB Benchmark (2004) *Matlab Performance Tests*. The MathWorks, Inc., Natick, MA, Available at http://www.mathworks.com/.