RC22899 (W0309-067) September 12, 2003
Computer Science

IBM Research Report

Adaptive Memory Paging for Efficient Gang Scheduling of Parallel Applications

Kyung Dong Ryu, Nimish Pachapurkar
Arizona State University
Tempe, AZ 85287

Liana L. Fong
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Adaptive Memory Paging for Efficient Gang Scheduling of Parallel Applications

Kyung Dong Ryu   Nimish Pachapurkar   Liana Fong

Department of Computer Science & Engineering
Arizona State University
Tempe, AZ 85287

{kdryu, nimishp}@asu.edu

IBM T.J. Watson Research Center
P O Box 704
Yorktown Heights NY 10533

llfong@us.ibm.com

Abstract

Gang scheduling paradigm allows timesharing of computing nodes by multiple parallel applications and supports the coordinated context switches of these applications. It can improve system responsiveness and resource utilization. However, the memory paging overhead incurred during context switches can be expensive and may diminish the positive effects of gang scheduling. This paper investigates the reduction of paging overhead in gang scheduling environments by applying a set of adaptive paging techniques: selective page-out, aggressive page-out, adaptive page-in and background writing. Our experiments with NAS NPB2 benchmark programs show that these new adaptive paging mechanisms can reduce the job switching time significantly (up to 90%).

Index Terms – virtual memory, adaptive operating system, gang scheduling, parallel computing

1 Introduction

Gang scheduling is a parallel job-scheduling paradigm, which allows timesharing of computing nodes by multiple parallel jobs, and supports the coordinated context switch of scheduled jobs across many computing nodes. Among the prominent advantages of gang scheduling are an improved system response under mixed workloads, and better system utilization.

Many system implementations attempt to exploit the applicability of gang scheduling to avail of the aforementioned advantages [1,2,3]. A number of gang scheduling studies explored the performance characteristics of various scheduling structures and algorithms [2,4,5]. Invariably, the context switch overhead due to paging for over-committed memory is cited as a major performance consideration of gang scheduling [2,3,5]. For example, Moreira, et al reported the paging overhead results after running three instances of a job with a 45MB footprint on AIX systems with 128MB and 256MB memory [3]. The average execution time on the 128MB system for the three jobs was 3.5 times greater than the 256MB system.

In this paper, we present the design and development of an adaptive-paging prototype used to explore the possible reduction of paging overhead in gang scheduling environment. Block paging, also known as swap paging or pre-paging, is a set of techniques that can group together a working set of a user or a job into a block of pages. Such a block can then be transferred in and out of the paging device in a single I/O transaction. Ideally, a block of pages can be written out to contiguous sectors on paging devices. For non-parallel commercial systems with mixed interactive and batch workloads, block paging has been shown to improve the system response time and throughput due to a reduction in the number of I/O requests. Latency of the disk arm movement is the largest component of the time required to transfer data to and from the disk during paging. Grouping of many I/O requests in form of a single block reduces the total seek time of the disk [6]. However, there is no previous work that examines the effect of such techniques on scientific parallel environments.

Our adaptive paging, in contrast to block paging, is a set of mechanisms that adapt the memory paging in a system to minimize the I/O for gang scheduled applications. These mechanisms exploit the knowledge of gang schedules of the jobs and their observed behaviors to decide when, and which pages to swap in or out of memory. Adaptive paging can potentially be a very effective technique for improving the performance of gang scheduled parallel applications. It could both reduce the time for paging, and help aggregate all the paging required for a job switch at the beginning of the time quantum. This makes paging occur simultaneously over all nodes and facilitates the synchronization of computation among parallel nodes.

Even if the cost of memory is dropping and available physical memory size is increasing rapidly, memory is still a scarce resource because the size of applications is also growing at an even faster pace. Addressing this issue in the context of high performance parallel applications is more critical especially when interactive response time is desired and more than one jobs have to be admitted by over-committing the available memory.

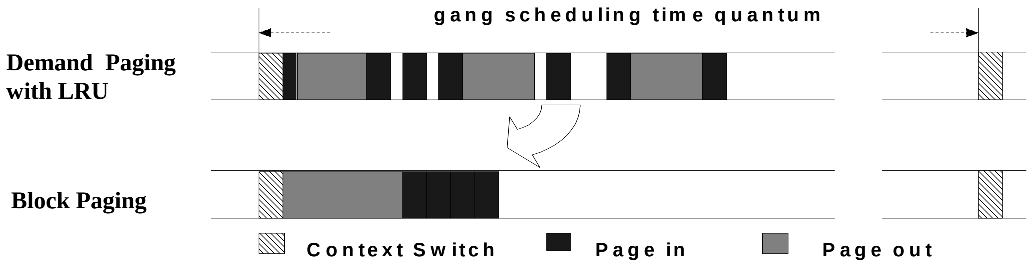

Figure 1: Memory paging compaction with adaptive paging

Paging large jobs in and out of memory causes large delays in job execution time because the disk I/O speed is slower by over an order of a magnitude than the CPU speed. Our prototype was designed to reduce the paging overhead by exploiting both the efficiency of adaptive paging techniques and the characteristic memory usage pattern of gang scheduled parallel applications. We built the prototype on a Linux kernel, and performed a set of experiments to quantify the reduction of paging overhead in gang scheduling environments.

The rest of this paper is organized as follows. Section 2 describes the memory paging model and its effect on gang scheduling. Section 3 describes the design and implementation of a set of adaptive paging mechanisms. Section 4 reports on the results of the experiments with gang scheduled NAS NPB2 benchmark programs. Section 5 reviews related work and Section 6 concludes with a summary and future work.

2 Memory Paging and Gang Scheduling

Modern operating systems can run multiple processes with large memory requirements concurrently, using virtual memory. Memory pages are placed in physical memory only when they are demanded (demand paging). When the available free memory is below a certain threshold, kernel must swap some in-memory pages out to the secondary storage. Such victim pages are selected using the Least Recently Used (LRU) algorithm or one of its approximations.

For page replacement, many operating systems including Linux adopt the clock algorithm – an approximation of LRU– where, instead of keeping track of the age of every page, a binary reference bit is used [21]. Swap-out operations start when free memory goes below a certain threshold – *freepages.min*. In Linux, the kernel examines the process that has the largest memory size and sweeps through it to select a group of non-referenced pages for eviction[1]. This action continues until the free memory in the system reaches the predefined upper limit – *freepages.high*. This watermark style page-out model is common for most of the Unix systems.

[1] The Linux kernel version 2.2

One key issue concerning paging is that frequent page faults have a negative impact on the system performance. The frequent disk I/O causes the process to remain idle for a considerable amount of time. This process is more efficient when done in blocks of multiple pages. Bundling multiple consecutive pages from disk for page swapping can amortize the seek time.

Though LRU is a widely accepted page replacement policy for general-purpose operating systems, it can be ineffective when multiple processes are gang scheduled for time-sharing of parallel machines. In the following paragraph, we describe the properties of gang scheduling of parallel applications, and the incompatible attributes with the current implementations of demand paging.

- **Large time quantum**: the time quantum for such applications is much larger than usual – in terms of minutes rather than tens of milliseconds. Due to a fair time-sharing based upon round robin scheduling, the process whose time quantum has just expired will not be scheduled for one or more time quanta.
- **Large working set**: the working set size of parallel applications is typically very large. When the time quantum of an application expires, its pages will not be evicted since they have been recently referenced.
- **Useful residual pages**: when a process is rescheduled, it is possible to discover that some of its pages have remained in memory since its last turn. However, when the process starts faulting for its evicted pages, The LRU policy often throws those residual pages out and brings them back again. This false eviction loses the opportunity to reuse the residual memory pages.

The above observations indicate that some additional scheduling information from the user-level gang scheduler can be very useful for more suitable memory paging in the kernel.

Figure 1 illustrates the behavior of memory paging in gang scheduling. When a job is awakened for its turn, after a context switch, it starts faulting for the pages that have been swapped out to disk by the previous job. However, the page faults (black bursts in the figure) are scattered over the scheduling quanta because pages are brought in only when the process attempts to access them. Moreover, page-out activities (gray bursts) further interleaves the page-ins. Therefore, many short computation bursts (white spaces) are interspersed with disk I/O. As shown in the second diagram in Figure 1, compaction of the frequent paging I/Os can reduce the job switching time.

To implement this idea, we will propose a set of adaptive paging mechanisms in the next section. We believe, however, that intra-job paging activities, which are not related to gang scheduled job switching, should be handled by the original paging policy. Thus, we try to keep the changes to the operating system kernel minimal and make them effective only in the job switching phase of gang scheduling; the details of implementation strategies are described in Section 3.5.

3 Adaptive Paging Mechanisms

In gang scheduling, unlike time-sharing of interactive serial applications, the scheduling of processes is usually controlled by an external user-level scheduler. The following information, supplied by a gang scheduler, can be very useful for memory page replacement

1. Which process is scheduled and which process is de-scheduled?
2. How large is the working set of the process being scheduled?

This information can signal the kernel as to which pages will be used soon and which pages will not be accessed for a long time. It is due to the fact that the memory pages of the process currently being de-scheduled – this process will be referred to as the *outgoing process* – will not be accessed until the next gang scheduling turn and the working set of the process being currently scheduled – this process will be referred to as the *incoming process* – will be accessed in a short time.

Therefore, instantly swapping out all the pages of the outgoing process in blocks and swapping in the whole working set of the incoming process in blocks will speed up the disk accesses and page handling. The working set sizes of the two processes would determine the number of blocks in the swapping action. To support effective adaptive paging for gang scheduling, we propose four paging mechanisms, *selective paging out, aggressive paging out, adaptive paging in* and *background writing*.

3.1 Selective Paging Out

From the experiments with gang scheduled parallel applications, we observed that LRU sometimes evicts memory pages that are soon to be accessed. This *false eviction* occurs as follows. Consider two gang scheduled parallel jobs, A and B, whose processes at each node have a large memory footprint. The processes of A will execute at all the nodes in gang for its time quantum; the quantum length is selected long enough to amortize the delay for fetching the typically large working set of the process into memory. When B is gang scheduled, each node causes a sequence of page faults and brings B's working set into memory. Due to memory shortage, A's pages in memory are swapped out to make room for B's working set. Nonetheless, some of A's pages may remain in memory since the working set of an individual process is typically smaller than physical memory. In its next turn, A is rescheduled and starts generating page faults. This time, A's lingering pages from the last turn will be swapped out first, because they are older than B's pages. However, the lingering pages – though not recently used – will soon be brought back in since they were a part of the working set of A.

```
try_to_free_pages(out_pid)
      while(free_pages < freepages.high)
            p = (process with out_pid);
            if(resident memory size of p > 0)
                  select oldest page of p and reclaim its page frame;
                  add the frame to freepages;
            else
                  use default page replacement technique (LRU);
            end if
      end while
end
```

Figure 2: Selective page-out algorithm

```
aggressive_page_out(out_pid, in_pid)
      pin = (process with in_pid);
      pout = (process with out_pid);
      target_free = VM size of pin (from last quantum);
      while(free_pages < freepages.high + target_free)
            page out a page belonging to pout;
      end while
end
```

Figure 3: Aggressive page-out algorithm

Our proposed selective page-out algorithm is to prevent the false eviction. The kernel paging module is now supplied with the outgoing process's ID. Whenever it selects victim pages for swap out, it first examines the pages of an outgoing process in the order of decreasing age. It considers the pages of other processes only when all the pages of the outgoing process are swapped out. The algorithm is summarized in Figure 2.

In most cases with gang scheduled applications, physical memory is larger than the working set size of any of the gang scheduled processes but it is not large enough to keep the working sets of all the processes. The selective page-out algorithm avails of this opportunity to reduce the paging overhead.

3.2 Aggressive Paging Out

Another inefficiency of memory paging under gang scheduling comes from the granularity of page memory swapping. Page replacement modules in most Unix-based operating systems swap out memory pages when a requested page is not found in the physical memory and there is not enough free memory. Thus, each page fault to bring in the working set will be slowed by the scattered page-out activities to make room in the physical memory.

Instantaneously making enough room for the working set of the incoming process will eliminate the interruptive scattered paging-out activities (previously illustrated in Figure 1). At the job switch in gang scheduling, our aggressive page-out module obtains the working set size of the incoming process and aggressively pages out the outgoing process until there are enough free pages available for the working set. The subsequent page faults will not cause any page-outs and the resulting disk I/O will be handled more efficiently. The kernel obtains the working set size using the page references during the incoming process' previous time quanta. Figure 3 summarizes this algorithm.

3.3 Adaptive Paging In

Swapping in memory pages one by one at each page fault is very expensive. Performing disk I/O in blocks is usually better since it reduces the effect of disk latency. Thus, at each page fault the original Linux page replacement reads ahead a group of subsequent pages. The default group size in the Linux kernel version 2.2 is 16. For the default page size of 4 Kbytes, it is only 64 Kbytes. This small group size originated from the fact that it is usually undesirable to swap out too many pages to bring in pages that may not be useful at all. Also, disk I/O for read-ahead of too many pages will delay the time that the faulted page gets available.

However, a larger read-ahead size can be a benefit at job switches in gang scheduling. It will not page out any useful pages because only the pages of the outgoing process will be swapped out. Furthermore, the number of page faults will dramatically decrease due to a larger read-ahead. It will also make the disk I/O more efficient by minimizing the disk arm movement. Further, since the extra pages brought in might not be used at all, boosting the read-ahead size might actually degrade the performance.

Instead, we have devised a mechanism that records the process id and the pages of that process as they are flushed out from the memory at a job switch. Later, when the same process is scheduled again, page faults are induced artificially to bring these recorded pages into memory. This approach proves to be an effective and low-overhead mechanism to make the entire working set of the process available at the start of the scheduling quantum.

```
// During Page-out
      pout = (process with out pid);
      pg_rec = (base address of page list for pout);
      addr = (base address of page being flushed out);
      append the addr to the list;

// During page-in
      pin = (process with in_pid);
      pg_rec = (base address of page list for process pin);
      while(pg_rec.next != null)
            fault for page with base = pg_rec.addr;
            while(pg_rec.offset > 0)
                  fault for page with base = pg_rec.addr++;
                  pg_rec->offset --;
            end while
            pg_rec = pg_rec.next;
            delete old pg_rec;
      end while
```

Figure 4: Adaptive page-in algorithm

Since many of the pages referred to by a process are contiguous, our page recording module records just the offset as the number of contiguous pages from a given page address, thereby saving substantial amount of kernel memory needed for storing the recorded information. The algorithm is shown in Figure 4.

3.4 Background Writing of Dirty Pages

The paging performance at job switches can be further improved if less time is spent for paging dirty pages out. This can be achieved by writing dirty pages to disk prior to actual job switches. To this end, we developed another adaptive mechanism that writes dirty pages in background at a lower priority while the job is running. It can reduce the number of pages written to the disk during the job switch.

The duration of background writing must be adjusted carefully to avoid writing of same pages repeatedly. This duration depends on memory access pattern, and working set size of the application, and amount of memory swapped during the switch. It is difficult to formalize the optimum amount of time for background writing. With some experimentation we have found that background writing for last 10% of the time quantum minimizes the repeated writing of pages and improves the performance of co-scheduling further by about 10%.

For our experimentation set up, we have extended the memory management module of Linux kernel. When a switch in the kernel is activated, the page swap daemon is woken up and assigned a lower priority. The special code inserted in the swap out routines of the kernel flushes dirty pages of the running job to disk. The background writing is switched off when the actual job switch begins.

3.5 Mechanisms and Interfaces

We chose the Linux kernel to implement our adaptive paging mechanisms for several reasons. Firstly, it is widely adopted for high performance parallel computing, partly due to the success of Beowulf clusters [7]. Secondly, the source code is open and widely available to develop the necessary modifications. Since many active Linux users build customized kernels, our mechanisms could easily be patched into existing installations. Thirdly, the virtual memory management in Linux is an exemplary implementation of most UNIX systems. Although we implemented the proposed algorithms in Linux, the techniques are general enough to be ported to other Unix-based operating systems that uses demand paging and read-ahead for virtual memory management.

Figure 5 illustrates our implementation architecture. The gang scheduler is a user-level process that stops and resumes application processes. At each context switch time in the scheduling table, the scheduler sends SIGSTOP signals to all the processes constituting the current job and SIGCONT signals to the processes of the job to be scheduled. The adaptive paging algorithms are implemented and incorporated in the Linux kernel. The API of our adaptive paging consists of two functions, `adaptive_page_in()`,and `adaptive_page_out()`, each with three arguments: incoming process ID, outgoing process ID, and working set size. The working set size also can be estimated by the kernel using the incoming process' run during the previous time quantum. In addition to these, there are

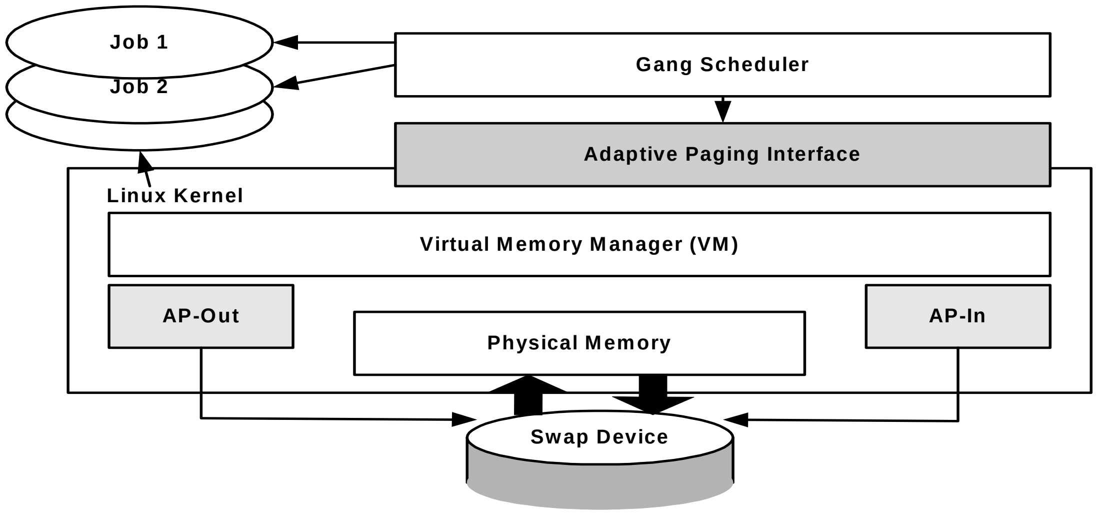

Figure 5: Implementation architecture for adaptive paging

two more functions: `start_bgwrite(inpid)` and `stop_bgwrite()` for activating and deactivating background writing. The communication between the user-level API and the kernel uses event notification by passing parameters through `/dev/kmem.` Alternatively, it can be achieved by extending the system calls. However, we avoided this alternative for a better portability to future revisions of the Linux kernel.

4 Performance Evaluation

To evaluate our adaptive paging mechanisms, we conduct a series of experiments using the NAS NPB2 benchmark. The experimental cluster consists of five machines at present: one running gang scheduler and the other four running parallel jobs. These machines are connected through a 100 Mbps Ethernet switch. Each machine has 1Gigabytes main memory and runs our adaptive-paging capable Linux developed on top of the Linux kernel 2.2.19. The degree of parallelism is limited to four, at this point, to minimize the network complexity caused by process synchronizations[2].

We select five representative combinations of our adaptive paging policies: adaptive page in (*ai*), selective page-out (*so*), selective aggressive page-out (*so/ao*), selective aggressive page out with background writing (*so/ao/bg*) and selective aggressive page-out with adaptive page-in and background writing (*so/ao/ai/bg*). For parallel jobs, we use five NPB2 benchmark applications: `LU, SP, CG, IS` and `MG` with the data class A. Five minutes were chosen for the time quanta of gang scheduling. Quantum larger than this will tend to negate the benefits of improved response time due to co-scheduling.

Figure 6 presents the paging activity traces for the first 50 minutes during the execution of two gang scheduled `LU.C`s for different combinations of our adaptive paging mechanisms on four machines. We reduce the available memory to 350 MB by wiring down some memory using `mlock()`. This reduction is necessary because the NPB benchmark has only a discrete set of data sizes – the data class `C` of `LU` uses only 188Mbytes when running on 4 machines in parallel –, and we need to stress the memory as in a real situation. At each job switch, which occurs every 300 seconds in the current configuration, a series of paging disk I/Os takes place to bring the working set of the incoming process into memory. As illustrated in the first graph, with the original LRU policy, page-in activities are spread over a long period of time. In this graph, the overlapping of page-ins and page-outs indicates that they interfere with each other and that the disk I/O efficiency decreases. The trace depicts that the paging occurs at a lower rate for a longer duration and hence delays the computation.

The second graph in Figure 6 demonstrates that the selective paging policy (*so*) decreases both amount and duration of paging by preventing the incoming process' old working set from getting swapped out. The extended paging period seen in both these graphs is essentially due to the thrashing caused by page-ins and page-outs happening at the same time. With the selective aggressive page-out (*so/ao*), as illustrated in the third graph, the paging overhead is further reduced due to the increased intensity of page-outs. When all three adaptive paging policies are applied, both page-in and page-out activities are intensified and compacted result-

[2] We are currently experimenting with 8, and 16 machines each having 1GB memory and 2GHz Intel Pentium 4 CPU.

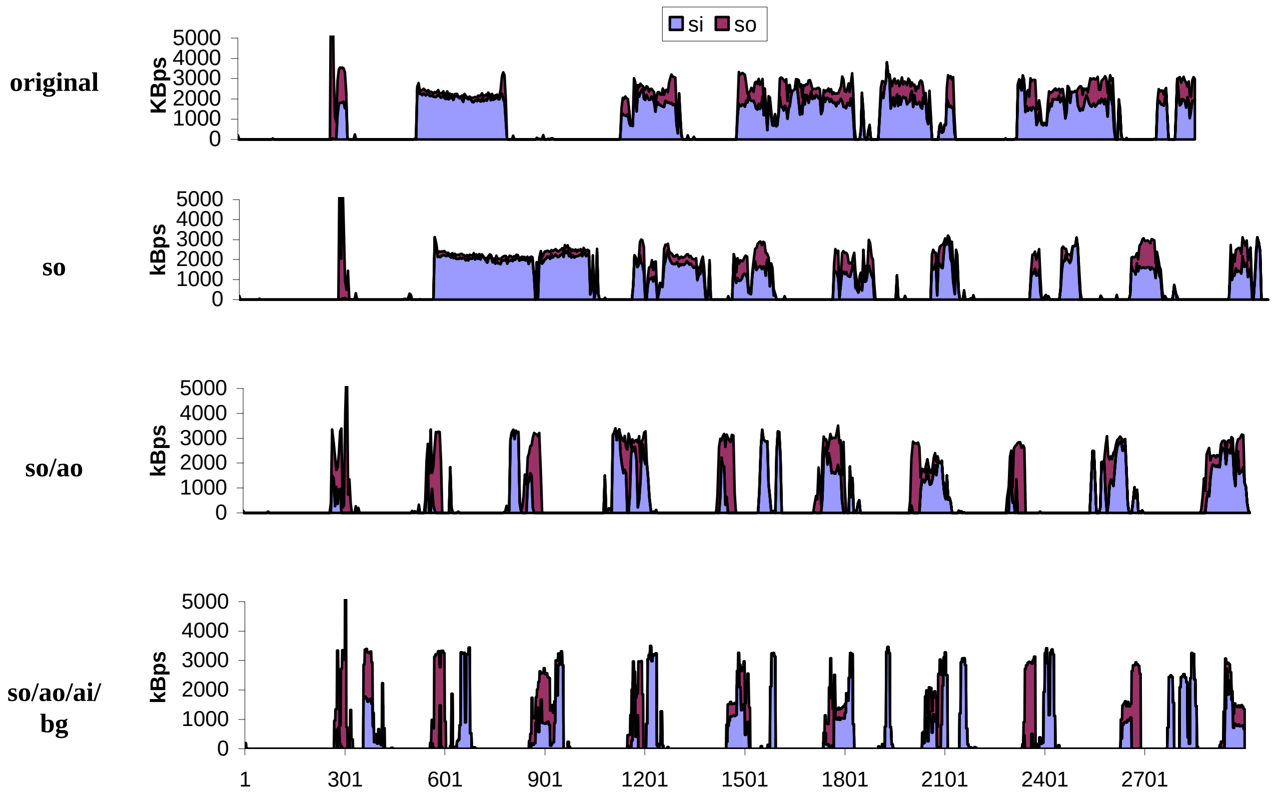

Figure 6: Paging activity traces of `LU` running on four machines for various adaptive paging policies.

ing in a more efficient memory paging. It is indicated by the sharp and high peaks in the fourth graph. Initial periods of page-in can be observed at some job switches because the background writing mechanism is active. Due to background writing, the page-out peaks during the switch have become shorter. Notice that the results follow closely the performance projection we illustrated in Figure 1. It verifies that our adaptive paging mechanisms effectively make memory paging in gang scheduling more compact and efficient. More experiments to quantify the benefits of adaptive paging are presented below.

4.1 Serial Jobs

To isolate the memory paging overhead from the network delay for parallel job synchronizations, we first run a serial version of NPB2 applications. Thus, the results demonstrate the effect of the efficient adaptive paging between two large memory applications running on a single machine. We use `LU, SP, CG, IS,` and `MG` with the data class `B`[3]. Two instances of each application run on a single machine using a gang scheduler with a five-minute time quanta.

Figure 7 summarizes the results of this experiment. Graph (a) plots the job completion time for the original LRU and the combination of all our adaptive paging policies (*so/ao/ai/bg*). The last set of bars, denoted by `batch,` is for the case when two applications run one after the other. The results show that the improvements vary depending on applications. The significance of the reduction in the completion time is in the order of `MG, LU, SP, CG,` and `IS`.

Using the batch completion time as a base, we computed the overhead imposed by job switches in gang scheduling, which is shown in Graph (b). This overhead indicates how much fraction of the time is spent on paging for job switching. For `SP, CG, IS,` and `MG,` the graph shows that the switching overhead is more than or close to 50% with the original paging algorithm, which means that more time has been spent for memory paging than the computation. To amortize this overhead, the time quanta should be increased sig-

[3] With this data class, the selected benchmark programs require 188MB to 400MB of memory which are suitable for our experimental setup.

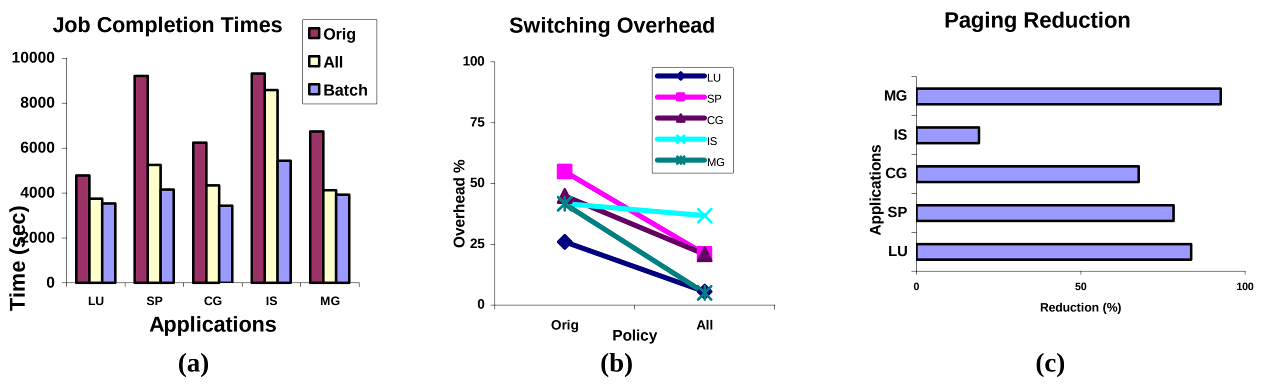

Figure 7: Job completion time, switching overhead and reduction for serial benchmarks

nificantly. However, an even larger time quanta would decrease the benefits of a better response and fairness in gang scheduling. The graph demonstrates that our adaptive paging policies reduce these overheads to between 5% and 37%. In case of `LU`, the overhead reduces from 26% to 5%.

Graph (c) presents the bars for the paging reduction over the original paging algorithm. For `MG`, the combination of all policies shows the biggest reduction, 93%. For `LU`, `SP`, and `CG`, the resulting reductions are 84%, 78% and 68%, respectively. The paging reduction of `IS`, is relatively small, 19%. This is due to its relatively small memory requirement. Overall, for the serial benchmark programs whose working size is large, our adaptive paging mechanisms were able to reduce the paging overhead by more than 65%.

4.2 Gang Scheduled Parallel Jobs

Now, we extend the experiments to the parallel versions of the NPB2 benchmark with the MPI communication library. Our adaptive paging mechanisms force the paging activities to occur simultaneously at all the

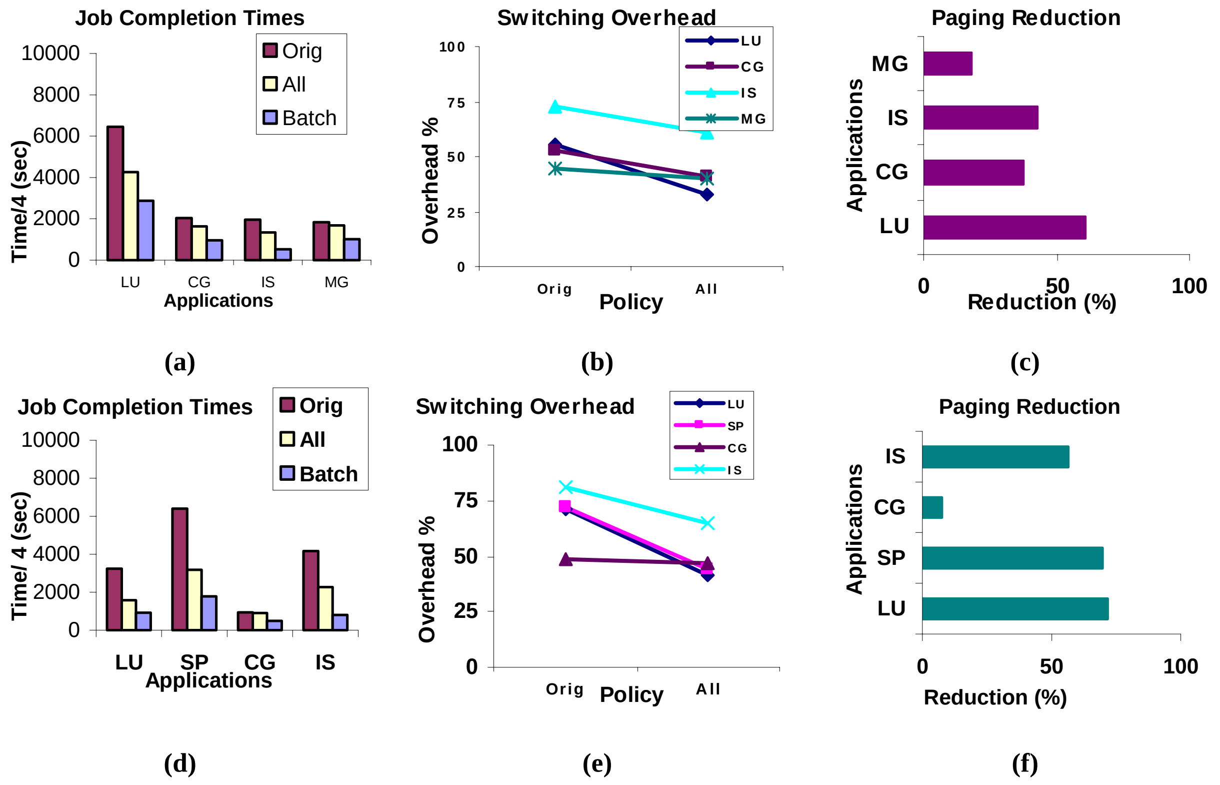

Figure 8: Job completion time, switching overhead, and reduction for parallel benchmarks (a-c: 2 machines; d-f: 4 machines)

nodes at the beginning of the context switch. We present the benefit of adaptive paging for the cluster-level gang scheduling.

In Figure 8, Graphs (a) and (d) demonstrates the job completion time for the four parallel benchmark applications running on two and four machines. Note that SP is included only for 4 machines since it does not compile for 2 machines, and MG is included only for 2 machines as its memory size is not suitable for our setup. *Batch* represents the base case for comparison when the two instances of each application run sequentially, one after the other incurring no paging for job switching. All the applications consistently improve the completion time with *so/ao/ai/bg*. `CG` typically has a small working set size and does not induce as much paging. It shows some improvement when run on 2 machines, but on 4 machines, its memory size and working set size reduce to such an extent that even with memory locking paging does not occur. Graph (d) shows that the reduction in completion time is minimum. `SP`, because of its large memory size, needs a longer quantum of 7 minutes to avoid continuous memory thrashing when run on 4 machines.

Graphs (c) and (f) illustrate the reduction in paging overhead over the original paging algorithm. `CG` manages a small reduction by 38% and 7% with *so/ao/ai/bg* policy when run on 2 and 4 machines, respectively. `LU` and `IS` reduce the job switching time by 61% and 72% for two machines, and 43% and 57% for four machines respectively when all three adaptive paging policies are used with background writing. `SP` shows significant reduction to 70% for four machines. This shows that our policies work even better when greater memory stress exists. The experiments with the parallel benchmark post a significant overall improvement in performance due to our adaptive paging algorithms.

In this section, we presented the results of the experiments with both the serial benchmark on a single node and the parallel benchmark with synchronization communications. We demonstrated that the memory paging time is a dominant factor for job switching and the use of our adaptive paging mechanisms can reduce it significantly. This reduction will enable the gang scheduler to use a smaller time quantum and hence to improve the responsiveness of parallel jobs.

4.3 Effects of Adaptive Paging Mechanisms

To better understand the effectiveness of each adaptive paging mechanism we developed, the detailed results of experiments are discussed in this section. Due to space limitation, only the results for LU are presented.

The graphs in Figure 9 show the results of serial and parallel runs (with both 2 and 4 machines). We have selected six representative combinations of our four mechanisms – *ai*, *so*, *so/ao*, *so/ao/bg*, and *so/ao/ai/bg*. Graph (a) shows the effect of various combinations of our paging policies on the completion time of the application. Note that measuring speedup from these results is not relevant since different input data sizes and memory locking sizes were used to emulate tight and over-committed memory by the gang scheduler. For both serial and parallel executions, *adaptive page-in* and *selective page-out* policies show the biggest reduction in completion time. Introduction of *aggressive page-out* reduces the benefit by a small amount in case of serial run because both the page-out policies together tend to cause too many page-outs. This negative effect is alleviated by *background writing* which disperses the page-outs over a slightly longer duration. For both psarallel runs, *aggressive page-out* actually helps *selective page-out* further by reducing the finish time by a small amount. Lastly, for all the runs, *adaptive page-in* reduces the completion time by reading in only the required pages.

For parallel runs, the original algorithm shows 55% to

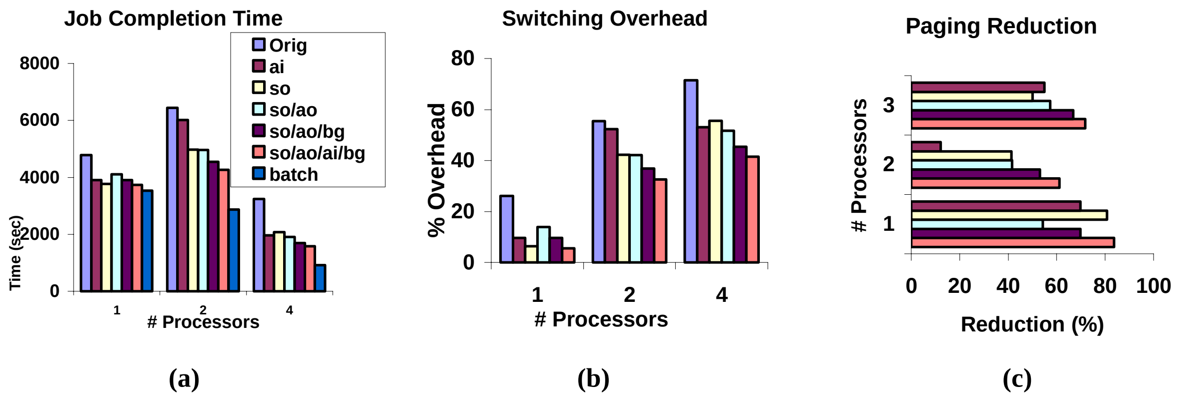

Figure 9. Detailed experimental results for `LU`

75% paging overhead as demonstrated in graph (b). The maximum reduction is achieved when all the policies work in cooperation. For parallel executions, the paging overhead is reduced to around 35% to 45% and with serial execution it is reduced to one-sixth of original algorithm.

Graph (c) shows the reduction in paging overhead achieved by using various combinations of mechanisms over the original algorithm. *Adaptive page-in* and *selective page-out* again prove to be the most effective strategies with more than 65% reduction in both cases. *Aggressive page-out* and *background writing* further help to reduce the overhead by 83%, 61% and 71% for serial, 2- and 4-machine executions, respectively.

5 Related Work

Many variations of gang scheduling systems for parallel applications have been proposed and experimented on [1,3,4,8,9,10]. Previous work exploring the context switch overhead for gang scheduling was limited to either reporting the observed performance impact of memory paging [3,4], or analyzing the possible impact on performance [5]. Wang, et al studied the relationship between the context switch overhead and the time quantum [11]. They suggested that it is better to use a longer time quantum for the systems with higher job switching overhead so that the overhead can be amortized over a longer execution time. However, the choice of longer time quantum would decrease the responsiveness of the system. This contrasts with the goal of gang scheduling to get good response time even for applications with higher memory demands.

A number of researches have been conducted on scheduling paradigms, in which memory is one of the resources to be scheduled in addition to the processing resources (processor cycles) [12,13,14]. Batat, et al found that exercising the admission control that allows only those jobs that fit into the available memory gives overall improvement in performance while suffering from delayed job execution [15]. Various algorithms aiming at advanced reservation of resources have been proposed and evaluated for computational grid environments [16]. However, the advantages of both of these approaches can only be realized if there is accurate a priori information on the memory utilization of jobs.

Zhang, et al analyzed the variations of page replacement implementations in recent Linux kernels (version 2.0, 2.2, and 2.4) to compare their abilities to deal with system thrashing [17]. They found that even though version 2.2 has relatively more effective protection against thrashing, none of these kernels have the ability to dynamically adapt to changes in memory demands; and, thus the protection that they provide is limited. This observation has a bearing on our choice of Linux kernel version 2.2 for implementing our modifications.

The dynamic resource allocation system, which is aimed at reducing the memory resource contention caused by page faults and I/O activity, was proposed by Xiao, et al [18]. In a cluster system with dynamic load sharing support, a small number of running jobs with unexpectedly large memory requirements may significantly increase the queuing delay times of the rest of the jobs with normal memory requirements. Chen, et al proposed a software method to deal with this problem of job blocking using virtual cluster reconfiguration mechanism [19].

Block paging is popular among many virtual memory systems [6,20]. However, none of them examines the effect of adaptive paging techniques on scientific environments. Wang, et al speculated that the block paging techniques used in non-parallel and timesharing systems may bring forth similar advantages to parallel applications [5].

6 Conclusion and Future Work

In this paper, we observed that the widely accepted demand paging model with the LRU page replacement algorithm does not exploit the characteristics of gang scheduled processes. We presented the adaptive paging mechanisms that were built on top of the Linux kernel and its performance impact on the gang scheduled applications.

Our design included the ideas of selective page-out, aggressive page-out, adaptive page-in, and background writing. Under this new adaptive paging scheme, the overall result showed a remarkable reduction of the paging cost in job switch for gang scheduling. Our experiments with NAS NPB2 benchmark programs showed that the job switching time can be reduced by up to 90%.

We are currently conducting experiments with a larger cluster with the NAS NPB2 benchmark and applications of various working set sizes. We are extending our performance study to parallel applications running on 8 and 16 nodes.

References

[1] A. Hori, H. Tezuka, Y. Ishikawa, et al. Implementation of Gang-scheduling on Workstation Cluster. Proceeding of IPPS Workshop on Job Scheduling Strategies for Parallel Processing, April, 1995.

[2] D. G. Feitelson, L. Rudolph. Evaluation of Design Choices for Gang Scheduling using Distributed Hierachical Control. Journal of Parallel and Distributed Computing, Vol 16, No 4, May, 1996

[3] J. E. Moreira, W. Chan, L. L. Fong, et al. An infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments. Proceeding of SC98. November, 1998.

[4] D. G. Feitelson, M. A. Jette. Improving Utilization and Responsiveness with Gang Scheduling. Proceeding of IPPS Workshop on Job Scheduling Strategies for Parallel Processing, April, 1997.

[5] F. Wang, M. Papaefthymiou, M. Squillante. Performance Evaluation of Gang Scheduling for Parallel and Distributed Multiprogramming. Proceeding of IPPS Workshop on Job Scheduling Strategies for Parallel Processing, April 1997.

[6] W. H. Tetzlaff, T. Beretvas, W. M. Buco, et al. A paging-swapping Prototype for VM/HPO. IBM System Journal, Vol. 26, No. 2, 1987.

[7] D. Ridge, D. Becker, P. Merkey, T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. Proceedings, IEEE Aerospace, 1997.

[8]G. Alverson, S. Kahan, R. Korry, et al. Scheduling on the Tera MTA. Proceeding of IPPS Workshop on Job Scheduling Strategies for Parallel Procoessing, April, 1995

[9] R. N. Lagerstrom, S. K. Gipp. PscheD-Political Scheduling on the Cray T3E. Proceeding of IPPS Workshop on Job Scheduling Strategies for Parallel Processing, April, 1997.

[10] E. W. Parsons, K. C. Sevcik. Implementing Multiprocessor Scheduling Disciplines. Proceeding of IPPS Workshop on Job Scheduling Strategies for Parallel Processing, April, 1997.

[11] F. Wang, H. Franke, M. Papaefthymiou, et al. A Gang Scheduling Design for Multiprogrammed Parallel Computing Environmnets. Proceeding of IPPS Workshop on Job Scheduling Strategies for Parallel Processing, April 1996.

[12] IBM/SP LoadLeveler product.

[13] V. G. J. Peris, M. S. Squillante, V. K. Naik. Analysis of the Impact of Memory in Distributed Parallel Processing System. Proceedings of ACM SIGMETRICS Conference, February, 1994

[14] S. K. Setia. The Interaction between Memory Allocation and Adaptive Partitioning in Message-Passing Multicomputers. Proceeding of IPPS Workshop on Job Scheduling Strategies for Parallel Processing, April 1997.

[15] A. Batat, D. G. Feitelson. Gang Scheduling with Memory Consideration. Proceeding of 14th International Parallel and Distributed Processing Symposium (IPDPS) 2000, May, 2000.

[16] W. Smith, I. Foster, V. Taylor. Scheduling with Advanced Reservations. Proceedings of IPDPS conference, May 2000.

[17] S. Jiang, Z. Zhang. Adaptive Page Replacement to Protect Thrashing in Linux. Proceedings of the 5th annual Linux showcase and conference, November 2001.

[18] L. Xiao, S. Chen, X. Zhang. Dynamic Cluster Resource Allocations for Jobs with Known and Unknown Memory Demands. IEEE transactions on parallel and distributed systems, March 2002.

[19] S. Chen, L. Xiao, X. Zhang. Adaptive and Virtual Reconfigurations for Effective Dynamic Job Scheduling in Cluster Systems. Proceedings of the 22nd international conference on distributed computing systems, 2002.

[20] S. J. Leffler, M, K. McKusick, M. J. Karels, J. S. Quarterman. The Design and Implementation of the 4.3.BSD Unix Operating System. Addison-Wesley Publishing Company.

[21] M. Beck, H. Bohme, M. Dziadzka, et al. Linux Kernel Internals. Addison-Wesley, 1998.