Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Future Generation Computer Systems 19 (2003) 505–519

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

www.elsevier.com/locate/future

Adaptive scheduling under memory constraints on non-dedicated computational farms

Dimitrios S. Nikolopoulos [a,*], Constantine D. Polychronopoulos [b]

[a] Department of Computer Science, The College of William and Mary, McGlothlin Street Hall, Williamsburg, VA 23187-8795, USA
[b] Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL 61801, USA

Abstract

This paper presents scheduler extensions that enable better adaptation of parallel programs to the execution conditions of non-dedicated computational farms with limited memory resources. The purpose of the techniques is to prevent thrashing and co-schedule communicating threads, using two disjoint, yet cooperating extensions to the kernel scheduler. A thrashing prevention module enables memory-bound programs to adapt to memory shortage, via suspending their threads at selected points of execution. Thread suspension is used so that memory is not over-committed by parallel jobs—which are assumed to be running as guests on the nodes of the computational farm—at memory allocation points. In the event of thrashing, parallel jobs are the first to release memory and help local resident jobs make progress. Adaptation is implemented using a shared-memory interface in the `/proc` filesystem and upcalls from the kernel to the user space. On an orthogonal axis, co-scheduling is implemented in the kernel with a heuristic that boosts periodically the priority of communicating threads.

Using experiments on a cluster of workstations, we show that when a guest parallel job competes with general-purpose interactive, I/O-intensive, or CPU and memory-intensive load on the nodes of the cluster, thrashing prevention reduces drastically the slowdown of the job at memory utilization levels of 20% or higher. The slowdown of parallel jobs is reduced by up to a factor of 7. Co-scheduling provides a limited performance improvement at memory utilization levels below 20%, but has no significant effect at higher memory utilization levels.

Keywords: Clusters; Computational grids; Scheduling; Memory management

1. Introduction

Computational grids provide opportunities to harness vast amounts of CPU cycles, memory, and network bandwidth, to meet the ever-growing computational needs of high-end applications. The success of grid computing depends on the ability of system software to exploit idle intervals on shared resources. Grid users are confronted with a dynamically configured, virtual parallel computer, without guarantees on contiguous resource availability.

Currently operating grids such as Condor pools [7] enable resource sharing in the form of social contracts. The users of a grid are classified into *hosts* and *guests*. A guest can use a resource only if no host is logged in. From a macro-perspective, the availability of idle memory and CPU time on large pools of computational resources is good enough to provide guests with the illusion of a parallel computer with several hundreds of processors [1,2]. Unfortunately, the terms of social contracts prevent guests from fully

[*] Corresponding author. Tel.: +1-757-2217739; fax: +1-757-2211717.
E-mail addresses: dsn@cs.wm.edu (D.S. Nikolopoulos), cdp@csrd.uiuc.edu (C.D. Polychronopoulos).

0167-739X/03/$ – see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0167-739X(03)00031-1

exploiting the idle cycles and memory of computational resources. This happens because a significant fraction of idle time on each resource is available while hosts are actually using their computers. In order to utilize better the available resources of computational grids, it is necessary to provide users with mechanisms and policies that exploit fine-grain idle intervals, using better mechanisms for multiprogramming and time/space sharing of resources.

1.1. The problems addressed in this paper

This paper is concerned with the utilization of idle memory in non-dedicated computational farms. Utilizing memory effectively is important, because the cost of low memory utilization on modern memory hierarchies is prohibitive. Excessive memory consumption leads to thrashing, due to swapping of pages to secondary storage. Thrashing is hard to predict or control and has detrimental effects on performance.

The first part of the work presented in this paper addresses the problem of thrashing prevention on non-dedicated computational farms. In particular, we are concerned with the problem of preventing thrashing triggered from guest jobs that compete with host jobs. Our strategy for thrashing prevention is based on extensions to the kernel scheduler and a user-level scheduling module, which provides guest jobs with the mechanisms needed to adapt under high memory pressure. The proposed scheduling policy attempts to prevent thrashing by suspending dynamically guest processes which are likely to cause thrashing. Processes are suspended at memory allocation points and at points at which their page fault activity appears to surge due to memory pressure, while the size of their resident sets does not increase. Suspended processes are resumed by the kernel, as soon as the operating system detects that enough free memory is available to run at least one of these processes.

The second part of the work presented in this paper addresses the problem of preventing thrashing, while improving the scheduling of communicating jobs running on a virtual parallel machine. Traditionally, the coordinated scheduling of parallel tasks on multiprogrammed parallel systems is addressed with co-scheduling algorithms [14]. These algorithms improve job coordination in parallel job scheduling, by ensuring that the communicating threads of a job run simultaneously on different processors. Unfortunately, these algorithms assume that parallel jobs have enough resources to run with their working sets in memory. This is controlled by a front-end batch scheduler that dispatches a job for execution only when the system has enough free memory to run the job. The size of the memory footprint of jobs is assumed to be known, either via a hint from the user, or via a historical profile of previous executions of the same jobs.

The problems of coordinated scheduling and thrashing prevention have been addressed in independent contexts. It is known though that the effect of paging on the performance of parallel jobs can be catastrophic [5,17]. We address the two problems simultaneously, with two scheduler extensions that prevent thrashing and improve co-scheduling. The key for integrating these scheduling extensions is proper prioritization of the corresponding scheduling actions. Prioritization is based on the execution conditions of the program and the relative impact of paging and co-scheduling on performance.

1.2. Contributions of the paper

In this paper, we present an adaptive scheduling strategy that prevents thrashing, enables adaptation to memory pressure at arbitrary points of execution in a program, and takes into account the relative priorities of jobs. The thrashing prevention scheduling module is integrated with a dynamic co-scheduling module. The two modules may work in synergistic or antagonistic manners. Using an experimental approach, we find that co-scheduling must take priority over thrashing prevention only when memory utilization is low. Thrashing prevention should be enforced in all other cases, because the cost of paging is much higher than the cost of loosing coordination between communicating jobs.

To investigate the performance of the scheduling extensions, we use workloads with mixes of parallel and sequential jobs. We use three sequential workloads, an interactive workload, an I/O-intensive workload with short-lived processes and a CPU and memory-intensive workload with long-lived processes. We present experiments in which the parallel jobs run with either of two priorities. In the first case, parallel jobs are executed with guest priorities and sequential jobs are executed with host priorities. Guest priorities are always lower than host priorities. As long as there is at least one host job running in the system, no guest job is using CPU time. In the second case, parallel jobs and sequential jobs compete using dynamic priorities assigned from a UNIX-like time-sharing scheduler with multilevel feedback queues. The workloads include also a microbenchmark, which exercises different levels of memory pressure on the system.

The focal point of the experiments is the performance of the parallel jobs on a non-dedicated cluster of privately owned workstations running general-purpose workloads. We show that our scheduling extensions allow for graceful degradation of the performance of parallel jobs in such settings. The improvements stem mostly from the thrashing prevention strategy, which reduces the page fault rate of parallel jobs by an order of magnitude. The co-scheduling extension provides moderate performance improvements at low memory utilization levels (less than 20%), but has no effect at higher memory utilization levels. In fact, a job should not accumulate priority for the sake of co-scheduling, if the system operates under memory pressure. As a rule of thumb, to effectively combine thrashing prevention and co-scheduling, one must give priority to thrashing prevention if memory utilization exceeds 20% and priority to co-scheduling at lower memory utilization levels.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the thrashing prevention scheduling algorithm along with some implementation details. Section 4 outlines the dynamic co-scheduling extension we are using in our setting and discusses how thrashing prevention and co-scheduling can be combined under different execution conditions. In Section 5 we present results from our experiments. Section 6 concludes the paper.

2. Related work

The adverse effects of paging and thrashing on the performance of parallel jobs have been identified and quantified by Burger et al. [5] and Setia [17]. Both works concluded that paging has such detrimental effects, that it should be avoided in its entirety. This observation motivated the introduction of admission control algorithms that prevent paging in parallel job schedulers. Admission control algorithms for limiting memory consumption were proposed for both time sharing and space sharing schedulers by Parsons and Sevcik [15], McCann and Zahorjan [10], and Batat and Feitelson [4], among others. The primary limitation of these algorithms is that they restrict the execution of jobs that do not fit in memory, based on a priori estimations of the sizes of the resident sets. In this work, we are proposing more flexible algorithms that detect and react to memory pressure while a job is actually executing. Furthermore, we investigate scheduling algorithms applicable in a general-purpose computing domain, as is the case with computational grids.

Jiang and Zhang [9] proposed a thrashing prevention mechanism implemented in the memory management module of the Linux kernel. This work shares similar motivation with ours. It proposes an adaptive thrashing protection heuristic, which avoids reclaiming pages from selected jobs, so that these jobs can run to completion and release memory resources. We propose a similar heuristic, which is however enforced by the jobs themselves rather than the kernel. Jobs detect the likelihood of thrashing and suspend their threads, thus avoiding touching memory until memory pressure is alleviated. The intuition is that resident jobs can execute faster without paging, while suspended jobs can anticipate a release of memory resources sooner if resident jobs execute faster. Furthermore, we investigate the problem of throttling memory consumption while improving the scheduling of parallel jobs.

Coordinated scheduling of threads of parallel jobs was originally proposed by Ousterhout [14]. Implicit co-scheduling algorithms based on local information appear to be the most appropriate forms of co-scheduling for loosely coupled parallel computing platforms, such as networks of workstations and grids. The co-scheduling algorithm proposed by Arpaci-Dusseau [3] utilizes information such as message round-trip times, to infer the scheduling status of communicating threads without using a centralized controller. Co-scheduling is enforced by adjusting the waiting times of threads at communication and synchronization points. Implicit co-scheduling is effective with simple communication patterns such as point-to-point communication. Unfortunately, it is very hard to find optimal waiting times for complex (e.g. all-to-all) and irregular communication patterns. Dynamic co-scheduling [18] addresses this problem to a certain extent, by using a combination of message volumes, frequencies, and sizes, to infer the scheduling status and the communication intensity of parallel jobs. The key idea of dynamic co-scheduling is to boost the priority of threads with high communication frequency and/or volume.

To the best of our knowledge, this work is the first to investigate the integration of scheduling methods for user-level adaptation of parallel jobs to non-dedicated computational farms, under memory constraints. A recent work by Giné et al. [8] addresses the same problems entirely in the operating system. This work presents a kernel scheduling heuristic which incorporates the paging rate and the communication frequency in the formula that calculates the scheduling priority of threads. Our approach differs in that adaptation is enforced at the application level and that our scheduling heuristic is designed to prevent thrashing, rather than remedy thrashing after its occurrence. In addition to that, we propose scheduling extensions which are less intrusive for non-parallel jobs running locally on computational nodes.

Recent work by Mills et al. [11] and Nikolopoulos and Polychronopoulos [12,13] has shown that it is possible to implement adaptive programs that react to the lack of sufficient memory by backing of, or rescheduling the computation. This paper advances this work to integrate adaptive scheduling for thrashing prevention with coordinated scheduling for better communication performance on non-dedicated environments.

3. Thrashing prevention algorithm

In this work we use a simple form of adaptation to memory pressure, which amounts to suspending the threads of the program at specific execution points, beyond which the program will cause thrashing, or at which thrashing has already begun as a consequence of the memory management policy of the operating system. A job keeps backing off its computation by suspending its threads, as long as memory pressure on the local node persists. This mechanism is implemented in a runtime library. The library interfaces with a kernel module that exports the information required to detect memory pressure.

The runtime library prevents programs from allocating and over-committing memory, by intercepting memory allocation points (including dynamic memory allocation and dynamic expansion of the address space in the kernel). The drawback of this strategy is that it might introduce starvation, if memory-resident jobs with very long execution times prevent other, potentially short jobs to acquire the necessary execution resources. In a general-purpose scheduling framework, this form of adaptation may hurt interactive jobs. In the context of this work, however, we consider computationally expensive jobs which are submitted to harness as many cycles as possible on a cluster or a computational grid. Since we restrict ourselves to this type of jobs, starvation may or may not be an issue. It is not an issue if we assume that these jobs run always at guest priorities, which are by definition lower than host priorities [16]. It may be an issue if we assume that these jobs compete with host jobs using standard time-sharing priorities. Our scheduler tolerates starvation for the latter type of jobs, via designating remotely submitted jobs as guests, without changing their dynamic priorities in the kernel. If a guest job has already allocated its working set, the user-level scheduling module uses the page fault rate and the size of the resident set of the job as indicators of thrashing. In that case, the threads of the job are suspended.

The proposed thrashing prevention algorithm detects memory pressure by checking two conditions. First, it checks if the allocated physical memory exceeds a threshold; second, it checks if the page fault rate[1] of a job exceeds a threshold, while the size of the resident set of the job does not increase. Both criteria are evaluated via hooks to the runtime library, which collects the necessary information from the kernel. The information is obtained with a shared-memory interface allocated by the kernel and mapped to the user space through the `/proc` filesystem. We describe the two conditions of memory pressure in more detail in the following sections.

3.1. *Detecting memory pressure using memory utilization thresholds*

In this case, programs evaluate memory pressure against their memory demands. To obtain estimates of the memory demands of programs, the user-level

[1] The term page faults refers to the so called *major page faults*, i.e. page faults which are serviced from the disk, rather than from a free list in memory.

library intercepts all memory allocation calls made by the program and records the requested amount of memory in a hash table shared between the user and the kernel through shared memory. This technique is not ultimately accurate as far as estimation of the working set of a job is concerned. It is used with the intention of preventing rather than resolving thrashing.

The algorithm is invoked from the programs and the kernel in an asymmetric manner. The programs invoke the algorithm at memory allocation points, when they detect that the amount of memory requested exceeds the amount of available physical memory. The invocation points can be statically identified in the runtime libraries linked to the programs. To circumvent the problem of reading and modifying system libraries, we are currently developing an alternative implementation based on dynamic instrumentation.

If a job detects that it is likely to thrash the system, it suspends its threads and puts them in a wait queue, until memory pressure is alleviated. These threads are resumed by the kernel, as soon as the kernel detects that the system's memory can accommodate the suspended job. In that case, the kernel puts the job in the run queue at the first scheduling opportunity. Multiple jobs that were previously suspended due to memory pressure are serviced on an FCFS basis, when they make attempts to acquire CPU time after suspension. This is done to avoid thrashing that might be caused by two or more simultaneously activated jobs which are competing for memory.

The anticipatory heuristic used in the algorithm assumes that in the common case, memory pressure is a short-lived event and guest jobs can wait for short local jobs to end and free up some memory space. This happens in many cases, such as interactive workloads. However, memory pressure may also be a long-term event, if the local jobs of a node run long CPU and memory-intensive programs. If guest programs are assumed to be running at priorities equal to the priorities of host programs, the problem of starvation of host and guest programs needs to be addressed. Our algorithm keeps guest programs suspended and lets host programs run and claim memory using their regular dynamic priorities. It is possible to use the suspension mechanism for hosts programs as well [13], however in this case, care must be taken so that at least one host program has sufficient memory resources to make progress. This can be easily resolved with selective page reclaiming algorithms, which prioritize the jobs from which they reclaim pages.

3.2. Detecting thrashing with page fault rates

Even if guest jobs prevent thrashing by suspending their threads, thrashing remains a live threat on multiprogrammed hosts. It is possible that while a guest program runs on the host, the host's jobs over-commit memory. The incurred thrashing shall prevent both host and guest jobs from making progress. The approach that we take in this case is to favor host jobs by suspending guest jobs. The problem that has to be solved is the transparent detection of thrashing by guest jobs. We use a simple mechanism that tracks the page fault rate of guest jobs in the kernel. When the kernel detects that the page fault rate of a guest job is increasing, it upcalls to the user-level scheduling module. The runtime library evaluates the following condition: If the page fault rate of the job increases for several consecutive time intervals, while the size of the resident set of the job does not increase, then thrashing has occurred. In that case, the threads of the job are suspended so that memory pressure is decreased. Otherwise, the guest job goes on as usual. The rationale behind this heuristic is that a job that keeps paging for long without increasing its resident set does not make any progress.

3.3. Algorithm and implementation

Fig. 1 shows the pseudocode of the thrashing prevention algorithm. The implementation of the algorithm is split into three parts, a user-level library, a loadable kernel module and kernel extensions. The user-level library suspends threads upon detecting memory pressure, either by polling variables in shared memory, or when the kernel upcalls to the library. The library intercepts memory allocation calls (both static and dynamic) made by the program. The kernel provides the amount of free physical memory, the resident set size of the job and the total memory allocated to the job throughout its execution, using a pinned physical page which is shared between the kernel and the user library. The page is allocated by the kernel and mapped to the user-level address space via an entry in the `/proc` interface.

```
User Level:
    at (each memory allocation point):
        intercept allocation size;
        while (allocation size $<$ free memory) {
                suspend the threads of the job;
                put the job at the end of the suspend queue;
        }

    at (each kernel upcall):
        check resident set size of past $N$ upcalls;
        if (resident set size unchanged $\wedge$ resident set size $<$ job size) {
                suspend job's threads;
                put job at the end of the suspend queue;
        }

Kernel Level:
      at (every scheduling point):
          if (guest jobs in suspend queue $>$ 0){
             pick first job in the queue;
             if (allocation request $<$ free memory) {
                place job in ready queue using the job's dynamic
                priority before suspension;
             }
          }

      for (each guest job): {
          check page fault rate during the last $N$ seconds;
          if (page fault rate $>$ threshold) {
             upcall to the user-level scheduler;
          }
      }
```

Fig. 1. User-level and kernel-level algorithms for adaptation under memory pressure.

A separate region of shared memory is used to store a queue of guest jobs which are suspended due to memory pressure. The queue is managed asymmetrically by the user-level library and the kernel. The user-level library inserts suspended threads in the queue, while the kernel wakes up suspended threads when memory pressure is alleviated.

The upcall mechanism works as follows. When the kernel detects that the job's page fault rate exceeds a threshold during the past N seconds, it resumes the program so that its execution proceeds inside the user-level library, rather than the originally preempted execution context. The kernel updates the resident set size of the job during the last N seconds in a table in shared memory. If the job detects that its resident set has not increased and the size of the resident set is less than the job's size, the job suspends its threads and puts them in the ready queue. Otherwise, the kernel scheduler is invoked and the original execution context of the job is restored.

The current prototype is implemented in Linux 2.4.18. The kernel changes amount to less than 100 lines of code without comments. The kernel module is approximately 500 lines of code and the user-level library is approximately 300 lines of code. The page fault threshold used in the implementation is 20 pages/s. The kernel updates job information (page fault rate and resident set size) in shared memory every second and the paramenter N is set to 2 s.

4. Co-scheduling

We selected a dynamic co-scheduling algorithm [18] and implemented it as an extension to the kernel scheduler. Dynamic co-scheduling algorithms achieve coordination by increasing the priority of jobs that receive or send messages over the network. The priority adjustment can be calculated by using parameters such as the number of messages, the size of messages, and the frequency of messages. In our implementation, the kernel increases periodically the priority of jobs, according to the number of packets that each process sends to or receives from the network. Note that the number of packets does not coincide with the number of messages. This heuristic views both jobs that send a small number of large messages and jobs that send a large number of small messages as communication-intensive jobs. Note also that the number of packets gives us a software-independent metric which can be easily tracked in the kernel (for TCP/IP and all other communication substrates for which the kernel provides buffering), or the device driver (for custom network interfaces).

Dynamic co-scheduling with priority boost is by far the simplest algorithm to implement. Although it does require modifications to the operating system, these modifications are minimal and straightforward. Our implementation of dynamic co-scheduling in Linux requires no more than 30 lines of code.

4.1. Combining co-scheduling and thrashing prevention

In theory, co-scheduling and thrashing prevention operate along orthogonal axes. The goal of co-scheduling is to ensure that two processes that need to communicate run at the same time on different processors. The thrashing prevention algorithm uses an anticipatory heuristic that tries to increase the throughput of a node by letting at least some memory-resident jobs proceed without suffering from thrashing.

Co-scheduling and thrashing prevention may work in synergistic or antagonistic manners under different conditions. Clearly, if a process that fits in memory needs to communicate with other processes, co-scheduling is beneficial. In the dual case, a non-communicating process that does not fit in memory should be suspended to prevent thrashing. The more involved case is the one in which a communicating process does not fit in memory.

If both co-scheduling and thrashing prevention are applied when a communicating job does not fit in memory, thrashing prevention takes priority over co-scheduling, because it enforces the suspension of the threads of the job, no matter what their priorities are at the moment. This means that the job is suspended even if it has not completed a communication step. The complementary strategy would be to let the job complete the communication step and then suspend it. Which of the two choices is the best depends on many things, such as the structure of communication and computation in the job, memory utilization at the time when the job needs to communicate, and the penalties of paging and uncoordinated scheduling.

Assume that a job needs to communicate and thrashing is about to prevail. If this job completes the communication step before the kernel starts reclaiming pages from the job, co-scheduling is likely to improve response time by helping the job and its peers proceed faster past the communication point. If the process is suspended by the thrashing prevention algorithm, co-scheduling and thrashing prevention are in conflict. This happens because co-scheduling attempts to raise the priority of a process which must be suspended to prevent thrashing. Which is the right decision depends on how the cost of paging compares to the earnings from co-scheduling. If scheduling the process is enforced despite memory pressure, the process might make progress but must pay the cost of paging. If the system prevents the job from running, it anticipates that memory will be released and the process will be able to run faster when it is rescheduled.

We strive for the latter option, i.e. giving priority to thrashing prevention over co-scheduling. A formal analysis might be applicable in this case, but our decision is driven by simple intuition and results from benchmarking. At high memory utilization levels, the penalty of paging exceeds by far the penalty of uncoordinated scheduling. Even on a lightly loaded system, dynamic co-scheduling does not buy us more than 20% of improvement of response time. On the other hand, thrashing can increase dramatically the response time of both communicating and non-communicating jobs. The undesirable effect of uncoordinated scheduling is primarily load imbalance, which can be mitigated in many ways at the application level. Thrashing is much harder to cope with at the application level.

4.2. Implementation issues

In the implementation, giving priority to thrashing prevention over co-scheduling is done as follows: the code that implements co-scheduling in the kernel refrains from increasing the priority of processes suspended due to memory pressure, by checking the shared queue of suspended processes. Co-scheduling is implemented by periodically increasing the priorities of communicating processes, whenever the kernel scheduler is invoked. Priority adjustments are calculated with the following formula:

$$b_t = \frac{1}{W}\frac{DL_{t-1} + L_t}{N+1} \qquad (1)$$

where b_t is the priority adjustment, L_t is the number of packets sent and/or received during period $(t-1, t)$, D is a decay factor and W is a normalization factor that adjusts b_t to the range of numerical values used for priorities in the kernel. The heuristic uses the recent history of message traffic to/from the process. The interval $(t-1, t)$ is set to 100 ms and the decay factor is set to 1. This means that the priority of each thread is increased according to the messages sent/received from the thread during the past 200 ms.

To avoid jeopardizing the performance of non-communicating processes, we modified the scheduler so that any extra time allocated to a communicating process due to a priority boost is taken back by the kernel as soon as it detects that the same process has no more messages to send or consume. With this modification, the accumulated CPU time awarded to each process is balanced in the long term and the native time sharing algorithm of the operating system is not significantly perturbed.

5. Experimental setting and results

Our experimental setting uses workloads that evaluate the effect of memory pressure on parallel jobs submitted to a cluster of non-dedicated workstations. The workstations in the cluster run different types of host workloads, including interactive workloads, I/O-intensive workloads with short-lived processes and CPU and memory-intensive workloads with long-lived processes. We are primarily interested in assessing the impact of having the parallel job competing with the host load on each node.

5.1. Scenario

The scenario of our experiments is outlined as follows. Each workstation in the cluster runs contiguously a host workload which can be any of the following:

- An interactive multiuser workload, which is a modified version of the *shared* workload of Caldera's AIM v. 7 benchmark suite [6]. This workload models multiple users working on a shared server and performing typical tasks such as editing text, browsing the web, working on spreadsheets and so on. We modified the workload so that the resident set size of the jobs varies between 1 and 10 MB.
- An I/O intensive workload with short-lived jobs. This workload is a modified version of the AIM *dbase* workload. The workload consists of a mix of 75% of I/O jobs that perform disks reads, writes and seeks and 25% of CPU-bound jobs that perform integer calculations on data and communicate through shared memory. We modified the workload so that the resident set size of the jobs varies between 10 and 50 MB.
- A CPU and memory intensive workload. This workload is a modified version of the AIM *compute* workload. More than 80% of the jobs in this workload perform intensive integer and floating point calculations. Around 5% perform disk I/O and the rest execute communication through shared memory and over the network. We modified the workload so that the sizes of the resident sets of the jobs vary between 50 and 100 MB. The workload is biased to model computationally expensive tasks with large memory footprints.

In the experiments, the workloads run contiguously on each node of the cluster for 6 h. While the workloads are executing, we launch another script which executes simultaneously two jobs:

- An MPI version of the NAS CG benchmark. The benchmark computes the smallest eigenvalue of a sparse matrix using the Conjugate-Gradient method. The application has an irregular communication pattern, which is a good stress test for co-scheduling algorithms. We used the Class A problem size. This problem size has a data set that occupies approximately 64 MB of memory per node.[2]

- A synthetic microbenchmark which allocates an array in memory and writes random bytes in each page occupied by the array. This microbenchmark is bound to run at the highest priority and is used to enforce a certain degree of memory utilization. The degree of memory utilization is the size of the array divided by the amount of physical memory available on each node.

CG is executed in two different modes. In the first mode, CG runs as a process with a special guest priority. The guest priority ensures that the threads of CG run only when there are no runnable jobs from the host workload on each node. The implementation of the guest priority is taken from [16]. The scheduler assigns a priority outside the standard UNIX priority range (−20, +20) and selects the thread from CG to run only when there is no process with a priority within the standard range. In the second mode, the threads of CG are assigned regular time-sharing priority.

We measure the mean slowdown of CG, calculated from the execution times of all instances of CG which are able to complete within the 6 h time frame that we set. The slowdown is obtained by dividing the mean execution time of the program while running with workload in the background, by the execution time of the program on idle nodes. We vary the level of memory utilization from 10 to 80% by adjusting the size of the array in the microbenchmark. Memory utilization beyond 80% could not be tested due to the limit imposed by the kernel on the amount of memory allocated to user programs.

The experiments were conducted on a cluster with four nodes. Each node is a Quad SMP with an Intel motherboard. It has four Pentium Pro processors clocked at 200 MHz, 512 KB of external L2 cache per processor and 256 MB of memory. The nodes communicate with TCP/IP over Fast Ethernet. The sequential execution time of CG in this cluster is 63 s, while the parallel execution time on four nodes with communication over Fast Ethernet is 13 s. Although the platform is quite outdated, it does not bias our results, because we select benchmarks with reasonable working sets. We used the newest version of the Linux kernel available at the time of the writing of this paper (Linux 2.4.18).

5.2. Results

Figs. 2–7 show the mean slowdown of CG with the different workloads. The labels on the charts correspond to the unmodified Linux kernel (BASE), the kernel with the scheduler patched with the dynamic co-scheduling extension (CO), the kernel patched with the thrashing prevention module (TP), the kernel with both the co-scheduling patch and the thrashing prevention patch, and thrashing prevention taking priority over co-scheduling (TP + CO) and the same setting, with co-scheduling taking priority over thrashing prevention (CO + TP).

As expected, increasing the memory demand of the workload increases the paging rate, which in turn slows down CG. Linux has a noticeable sensitivity to paging, even when the average memory utilization is as low as 20%.

At low levels of memory utilization (i.e. 10%), co-scheduling improves the performance of CG by a moderate factor, which ranges between 10 and 22%. On the other hand, the thrashing prevention algorithm incurs a noticeable runtime overhead. However, when memory utilization increases beyond the 20% threshold, the importance of thrashing prevention becomes clear. The kernel version that uses dynamic co-scheduling but no thrashing prevention shows the diminishing returns of the unmodified kernel, resulting in slowdowns that range between a factor of 7 and a factor of 150. Thrashing prevention reduces the slowdown of the non-co-scheduled parallel program by at least a factor of 3. In fact, thrashing prevention by itself is essentially as effective as the combination of thrashing prevention and co-scheduling, with two exceptions: the interactive workload when CG runs with guest priority; and the CPU and memory-intensive workload, when CG runs with guest and standard priorities.

[2] We conducted experiments using seven MPI benchmarks from the NAS suite and obtained very similar results. CG is the one that appears to be more sensitive to the loss of coordination between threads.

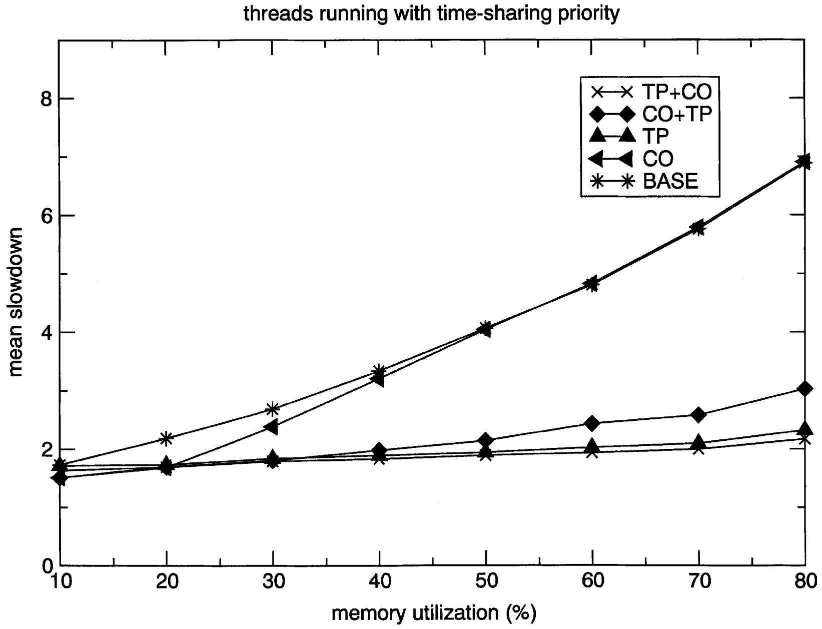

Fig. 2. Slowdown of CG, when the threads of CG compete with the interactive workload, at different levels of memory utilization.

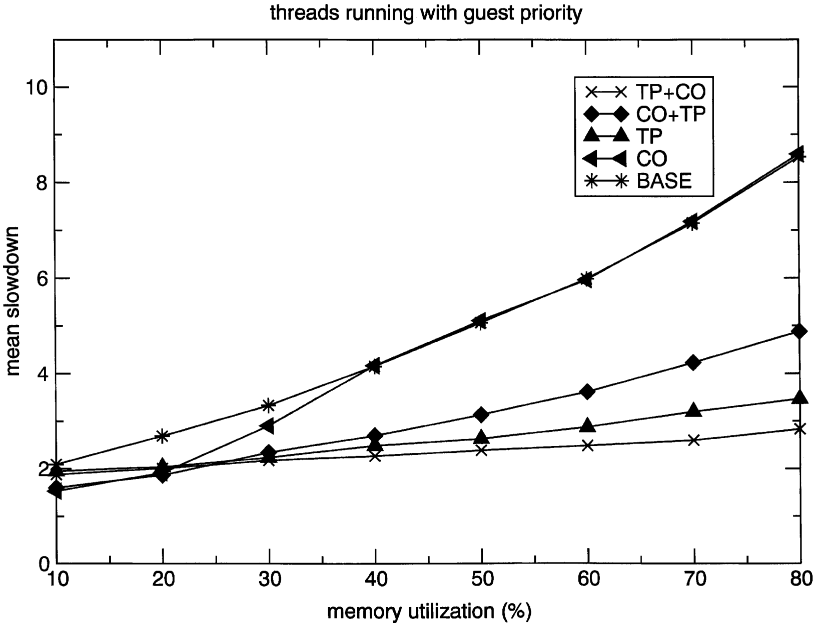

Fig. 3. Slowdown of CG, when the threads of CG run with guest priority and compete with the interactive workload, at different levels of memory utilization.

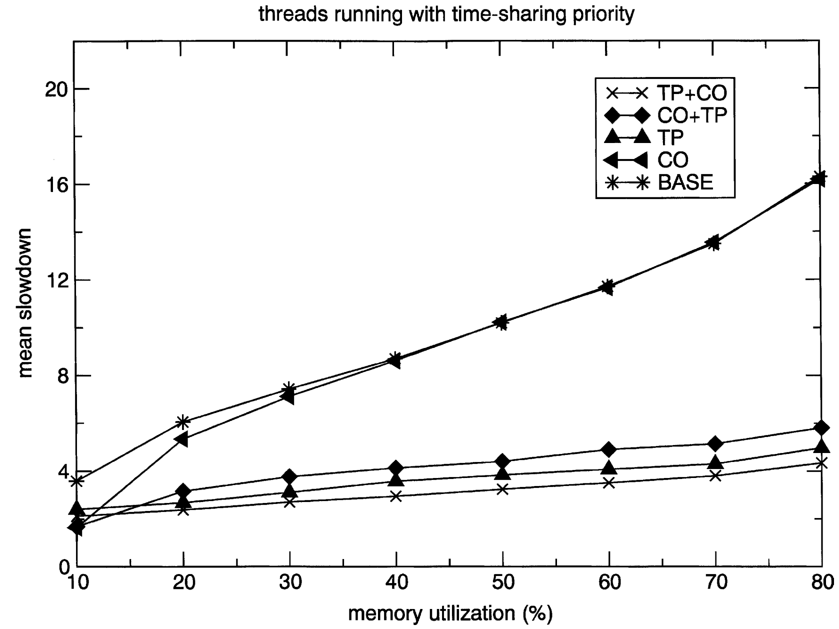

Fig. 4. Slowdown of CG, when the threads of CG compete with the I/O-intensive workload, at different levels of memory utilization.

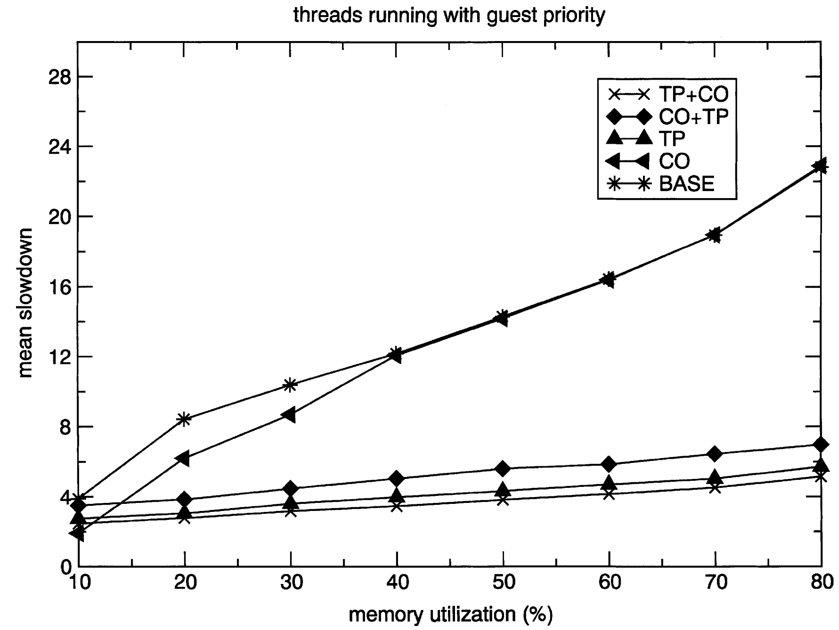

Fig. 5. Slowdown of CG, when the threads of CG run with guest priority and compete with the I/O-intensive workload, at different levels of memory utilization.

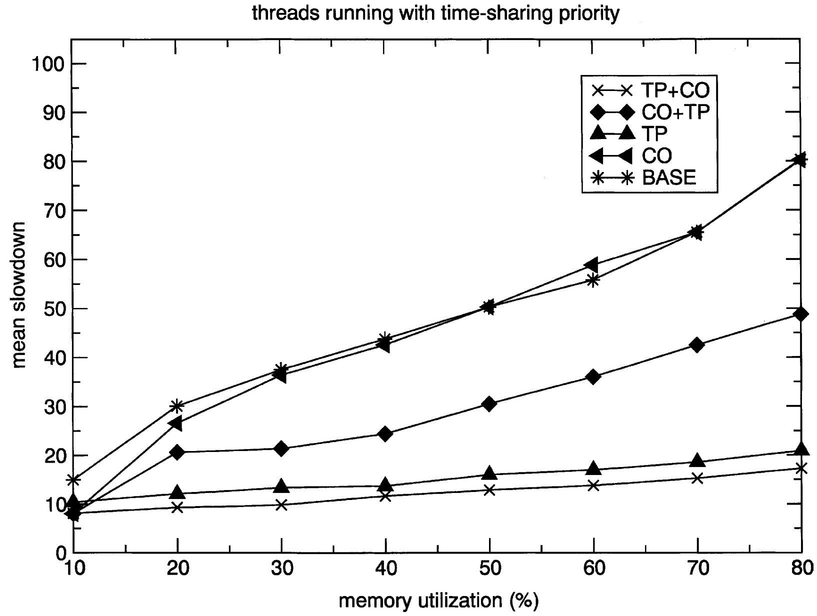

Fig. 6. Slowdown of CG, when the threads of CG compete with the CPU and memory-intensive workload, at different levels of memory utilization.

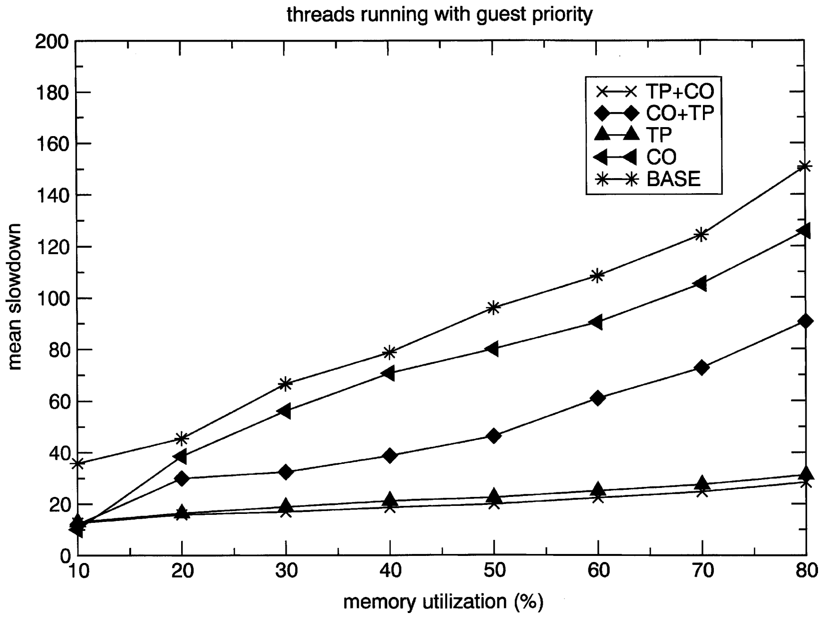

Fig. 7. Slowdown of CG, when the threads of CG run with guest priority and compete with the CPU and memory-intensive workload, at different levels of memory utilization.

The three workloads and the priority at which the parallel program runs have different impacts on performance. The interactive workload has minimal CPU and memory requirements. It slows down the parallel program by no more than a factor of 7. Thrashing prevention reduces the mean page fault rate of CG by a factor of 10 and the slowdown of the program by a factor of 3.3. The impact of running CG at guest priorities is moderate (a 12% increase of slowdown), because the interactive workload leaves plenty of idle intervals for both CPU time and memory. Co-scheduling reduces slowdown on up to 20% memory utilization.

The I/O-intensive workload has almost twice as much impact as the interactive workload on the parallel program. This is a result of the additional memory resources required by the I/O-intensive workload and a significant fraction of CPU time wasted in system activity to service I/O interrupts. Running the parallel job at guest priority adds at least 20% to the slowdown. Thrashing prevention reduces the mean page fault rate of CG by a factor of 19 and the mean slowdown of CG by at least a factor of 5. Co-scheduling reduces slowdown only up to 10% memory utilization.

The CPU and memory intensive workload has the most severe impact on performance. This workload occupies constantly the CPU and around 40% of the available physical memory. The nodes start thrashing when the synthetic microbenchmark uses as little as 20% of the available memory. Running the job at guest priorities increases the slowdown by another factor of 2, because the workload leaves very few opportunities to exploit idle CPU and memory. Thrashing prevention reduces the mean page fault rate of CG by a factor of 32 and the mean slowdown of CG by factors of 7 or more. Co-scheduling reduces slowdown only up to 10% memory utilization.

6. Conclusions

This paper presented scheduler extensions for efficient execution of parallel jobs on non-dedicated computational farms. The purpose of the extensions is to utilize better the idle memory resources of the nodes and allow for a non-intrusive exploitation of resources in privately owned environments. We have presented scheduling extensions that enable adaptation to memory pressure and co-scheduling of communicating threads. Adaptation is enabled by an interface that gives programs the ability to prevent thrashing via suspending their threads at specific points of execution. Computation is suspended dynamically, either at memory allocation points to prevent memory from being over-committed, or via upcalls from the kernel to the program, when the page fault rate and the resident set size of the program combined, indicate that the system is thrashing.

The paper has shown that thrashing prevention is more important than co-scheduling, when memory resources become scarce. At low memory utilization levels (20% or less) co-scheduling appears to be a useful extension that improves the performance of communication-intensive parallel programs by approximately 20%. At higher levels of memory utilization though, co-scheduling is essentially of no use, because the cost of paging dominates execution time.

There are many directions for future research in this area. Perhaps the most important one is to develop scheduling interfaces that can be exploited in application-specific manners. In this paper, we have assumed a generic scheduling interface that is oblivious of the actual status of the program when thrashing is about to occur. Much better scheduling and memory management decisions can be taken if the runtime system has knowledge of the criticality of the tasks in a program. The related problems need to be studied in the contexts of specific applications. Another issue we currently investigate is the extension of the interface to enable global rather than local adaptation of a program, with mechanisms such as job migration and adaptive control of the number of threads.

Acknowledgements

An earlier version of this paper appeared in the Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'02). We would like to thank the anonymous referees for many insightful comments that helped us improve the quality of the paper. This work was supported by the National Science Foundation grant No. ITR-00-85917, the Office of Naval Research grant No. N00014-96-1-0234, a research grant from the National Security Agency, and a research grant from Intel Corporation.

References

[1] A. Acharya, G. Edjlali, J. Saltz, The utility of exploiting idle workstations for parallel computation, in: Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'97), Seattle, WA, June 1997, pp. 225–236.

[2] A. Acharya, S. Setia, Availability and utility of idle memory in workstation clusters, in: Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99), Atlanta, GA, May 1999, pp. 35–46.

[3] A. Arpaci-Dusseau, Implicit coscheduling: coordinated scheduling with implicit information in distributed systems, ACM Trans. Comput. Syst. 19 (3) (2001) 283–331.

[4] A. Batat, D. Feitelson, Gang scheduling with memory considerations, in: Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS'2000), Cancun, Mexico, May 2000, pp. 109–114.

[5] D. Burger, R. Hyder, B. Miller, D. Wood, Paging tradeoffs in distributed-shared-memory multiprocessors, in: Proceedings of the IEEE/ACM Supercomputing'94: High Performance Networking and Computing Conference (SC'94), Washington, DC, November 1994, pp. 590–599.

[6] Caldera International Inc., AIM Independent Resource Benchmark, Suite VII, 2001.

[7] J. Frey, T. Tannenbaum, M. Livny, I. Foster, S. Tuecke, Condor-G: a computation management agent for multi-institutional grids, in: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10), San Francisco, CA, August 2001, pp. 55–63.

[8] F. Giné, F. Solsona, P. Hernandez, E. Luque, Coscheduling under memory constraints in a now environment, in: Proceedings of the Seventh Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'01), in conjunction with ACM SIGMETRICS'01, Boston, MA, June 2001.

[9] S. Jiang, X. Zhang, Adaptive page replacement to protect thrashing in Linux, in: Proceedings of the Fifth Annual Linux Showcase and Conference, Oakland, CA, November 2001.

[10] C. McCann, J. Zahorjan, Scheduling memory constrained jobs on distributed memory parallel computers, in: Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'95), Ottawa, Canada, May 1995, pp. 208–219.

[11] R. Mills, A. Stathopoulos, E. Smirni, Algorithmic modifications to the Jacobi–Davidson parallel eigensolver to dynamically balance external CPU and memory load, in: Proceedings of the 15th ACM International Conference on Supercomputing (ICS'2001), Sorrento, Italy, June 2001, pp. 454–463.

[12] D. Nikolopoulos, C. Polychronopoulos, Adaptive scheduling under memory pressure on multiprogrammed clusters, in: Proceedings of the Second IEEE/ACM International Conference on Cluster Computing and the Grid (ccGrid'02), Berlin, Germany, May 2002, pp. 22–29.

[13] D. Nikolopoulos, C. Polychronopoulos, Adaptive scheduling under memory pressure on multiprogrammed SMPs, in: Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS'2002), Fort Lauderdale, FL, April 2002.

[14] J. Ousterhout, Scheduling techniques for concurrent systems, in: Proceedings of the Third International Conference on Distributed Computing Systems (ICDCS'82), Miami, FL, October 1982, pp. 22–30.

[15] E. Parsons, K. Sevcik, Coordinated allocation of memory and processors in multiprocessors, in: Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'96), Philadelphia, PA, May 1996, pp. 57–67.

[16] K. Ryu, J. Hollingsworth, P. Keleher, Mechanisms and policies for supporting fine-grain cycle stealing, in: Proceedings of the 13th ACM International Conference on Supercomputing (ICS'99), Rhodes, Greece, June 1999, pp. 93–100.

[17] S. Setia, The interaction between memory allocation and adaptive partitioning in message-passing multicomputers, in: Proceedings of the First Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'95), in conjunction with IEEE IPPS'95, Lecture Notes in Computer Science, vol. 949, Santa Barbara, CA, April 1995, pp. 146–165.

[18] P. Sobalvarro, S. Pakin, W. Weihl, A. Chien, Dynamic coscheduling on workstation clusters, in: Proceedings of the Fourth Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'98), Lecture Notes in Computer Science, vol. 1459, Orlando, FL, April 1998, pp. 231–256.

Dimitrios S. Nikolopoulos is an Assistant Professor of Computer Science at the College of William and Mary. Prior to joining William and Mary, he was a Visiting Research Assistant Professor of the Coordinated Science Laboratory at the University of Illinois, Urbana-Champaign. He received his PhD in Computer Engineering in 2000 and the Diploma in Computer Engineering in 1996, both from the University of Patras, Greece. His research focuses on the integration of system software components for better adaptation of programs to high-end computing platforms. He has authored more than 35 technical publications in major international journals and conferences, including three publications that won Best Paper Awards.

Constantine D. Polychronopoulos is a Professor of Electrical and Computer Engineering and a Research Professor of the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. At CSL, he is leading research projects that focus on creating new hardware and software approaches to speed distributed computation. Specific research projects include the design and implementation of novel compiler optimizations for ILP and multithreading, automatic program parallelization, runtime systems and operating system cores for multithreaded and multiprocessor architectures, and multithreaded micro-architecture design and performance characterization. He received his PhD in Computer Science at the University of Illinois at Urbana-Champaign in 1986, his MS in Mathematics from Vanderbilt University in 1982, and his BSc in Mathematics from the University of Athens, Greece in 1980. His research work has appeared in more than 100 journal, conference and other technical publications, and has been funded by DARPA, ONR, NSA, NSF, and industrial labs. He was elected on the Board of Directors of ACM SIGARCH in 1990, and is the recipient of a Fulbright scholarship, the 1989 NSF Presidential Young Investigator Award, and the 1998 Bodossaki Foundation Award.