Adaptive Scheduling under Memory Pressure on Multiprogrammed Clusters*

Dimitrios S. Nikolopoulos and Constantine D. Polychronopoulos

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, IL 61801, U.S.A.
dsn@csrd.uiuc.edu, cdp@csrd.uiuc.edu

Abstract

This paper presents scheduling techniques that enable the adaptation of parallel programs to clustered computational farms with limited memory capacity. The purpose of the techniques is to coschedule communicating processes and prevent paging, using two cooperating extensions to the kernel scheduler. A paging prevention module enables memory-bound programs to adapt to memory shortage, by suspending their threads at well-defined execution points. The associated operating system interface provides a generic mechanism that enables programs to adapt in different ways, including application-specific forms of adaptation. At the same time, a dynamic coscheduling heuristic implemented in the kernel scheduler increases periodically the priority of communicating processes so that parallel jobs are eased through communication points. We show that when a parallel job competes with randomized sequential load running in the nodes of the cluster, the combination of coscheduling and paging prevention reduces drastically the slowdown of the job at high levels of memory utilization. We also show that if memory resources are ample, coscheduling should take priority over paging prevention, whereas if memory resources are scarce, preventing paging should take priority over coscheduling.

1. Introduction

An effective scheduling strategy for parallel and client-server applications on clusters with multiprogrammed nodes should ensure the coordinated allocation of processors to processes that need to communicate [3, 14]. In a different context, an effective memory management strategy for the nodes of the cluster and the cluster as a whole should try to avoid *paging*, i.e. the situation in which the operating system reclaims pages from running jobs, because the working sets of these jobs do not fit in the nodes' memories. So far, these two problems have been addressed independently. It is known though that the effect of paging on the performance of parallel jobs running on multiprogrammed systems can be catastrophic [16]. With few exceptions, the related literature attacks this problem with admission control mechanisms used in conjunction with specific scheduling and load balancing algorithms [4, 9, 15]. Admission control prevents a parallel job from executing, as long as the resident set of the job does not fit in the memory available to the job at submission time. The size of the resident set is assumed to be known, either via a hint from the programmer, or via a profile of a previous run of the program.

A different way to approach the problem of paging is to embed some form of adaptability to the programs, so that they can either prevent or react to paging by self-scheduling their computation. The idea of adapting to unexpected resource shortage becomes relevant in clusters which are not arbitrated by a front-end batch system, but allow transparent remote execution of parallel and sequential jobs. Such clusters can be found in several academic environments (including our own workspace) and their workloads are widely heterogeneous. Given the diversity of academic projects, these clusters may need to run simultaneously jobs such as MPI programs, cluster-enabled file systems, cluster-enabled Internet services, and sequential jobs submitted to harness idle CPU cycles and unutilized memory [1].

Recent work from the authors [13] has shown that with a simple extension to the operating system's interface, a job

*This work was supported by the National Science Foundation grant No. EIA-99-75019, 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.

Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02)
0-7695-1582-7/02 $17.00 © 2002 IEEE

running on a multiprogrammed server can easily detect the likelihood of paging at runtime and react accordingly. We have shown that this adaptive scheduling strategy can cope with the adverse effects of paging in SMPs running server workloads [13]. Unfortunately, in an attempt to utilize the same scheduling strategy in clusters, we found that one has to consider also the problem of coordinated allocation of processors to communicating processes.

In this paper, we show how adaptive local scheduling of jobs under memory pressure can be combined with dynamic coscheduling to improve the throughput of clustered farms, by alleviating the adverse effects of paging and uncoordinated scheduling simultaneously. Our basic assumption is that the target clusters are open to submissions of arbitrary job mixes, without explicit constraints about job types and sizes. We show that coscheduling and adaptive paging prevention can be combined so that coscheduling does not provoke paging and paging prevention does not harm coscheduling. Using a workload that mixes a single parallel job with randomized sequential load running in the background, we show that our scheduling strategy reduces drastically the slowdown of the parallel job. Neither coscheduling alone, nor paging prevention by itself are sufficient to achieve high throughput at all levels of memory utilization. Coscheduling appears to be more effective at low memory utilization levels, whereas paging prevention should definitely take priority over coscheduling at higher memory utilization levels. Effective combination of the two methods surpasses limitations in both cases.

To the best of our knowledge, this work is the first to investigate the integration of scheduling methods for the adaptation of parallel programs on multiprogrammed clusters with memory limitations. The most related paper that we are aware of [6] used a monolithic, kernel-level scheduling heuristic to deal with the same problem. Our work differs in that adaptation is enabled by the cooperation between the programs and the operating system, rather than the kernel scheduler on its own. We also provide a much more flexible interface for adaptation, which allows programs to use different adaptation techniques.

The rest of this paper is organized as follows. Section 2 outlines the motivation and provides the background and assumptions of this work. Section 3 describes our paging prevention mechanism. Section 4 shows how paging prevention and dynamic coscheduling can be effectively combined. In Section 5 we present the experimental setting and results. Section 6 concludes the paper.

2. Motivation and Background

Although a cluster employed to execute parallel jobs on dedicated processor sets might operate at satisfactory utilization levels with admission control from a front-end queuing system, an open computational farm with dynamic workloads is likely to benefit more from adaptive scheduling strategies. Parallel jobs that interfere with sequential jobs are extremely sensitive to load variation and resource scarcity. In particular, memory shortage, an artifact of running jobs with large memory footprints simultaneously, can slow down a process —and as a side-effect any other processes that communicate with that process— by as much as one order of magnitude. Unfortunately, the programs themselves can not predict load variations and the related research has shown that there is no single OS scheduling algorithm able to handle a heterogeneous workload without harming at least one class of jobs in the workload.

The thesis of this paper is that scheduling jobs when memory resources are limited and the load oscillates unpredictably should be performed in an adaptive manner. Furthermore, the required degree of adaptability should be embedded in the programs, rather than enforced by an overly complex scheduler in the operating system. This is a carefully thought decision. Although all scheduling mechanisms presented in this paper can be perfectly encapsulated in the operating system (and most of them are in the implementation of our prototype), the ultimate purpose is to let programs decide on the way they react to resource scarcity. This leaves the door open for algorithmic and application-specific optimizations for the adaptation of programs to multiprogrammed systems.

Different forms of adaptability can be considered in different cases. A well known form of program adaptability is rescheduling, as a response to variations of the number of processors allocated to the program. This form of adaptability can be enabled by mechanisms such as scheduler activations and kernel-user communication interfaces [2, 17]. Recent work has shown that it is possible to implement adaptive programs that react to the lack of sufficient memory by backing off or rescheduling the computation [10, 13]. Another form of adaptability which is suitable for clusters is process migration in the presence of either high CPU load or high memory pressure in individual nodes [5, 18]. The fact that we export the adaptability interface to the programs instead of hiding it in the operating system gives the flexibility needed to implement application-specific adaptability algorithms. We consider this as an important contribution of this work.

In our prototype, we use the simplest form of adaptability, which amounts to preempting and backing off the execution of the program as long as memory pressure in one or mode nodes of the cluster persists. This mechanism is generic, in the sense that it is applicable to any program, parallel or sequential, without requiring modifications to the source code. However, it is not a panacea. It deals mostly with the interference between a handful of memory-bound jobs and short-lived sequential jobs. This sort of interfer-

ence is not uncommon. It is encountered in clusters that run both parallel jobs and cluster-based services, such as distributed file systems or Internet services. The load from server applications is inherently random and bursty, including numerous short-lived sequential jobs with varying memory demands.

The purpose of the back-off mechanism is to let the programs themselves prevent paging. A program which detects that its resident set does not fit in memory suspends its threads[1] at runtime and anticipates that enough memory will be released soon, so that it can run at a higher speed. Parallel jobs with memory demands that push memory capacity to its limits at a specific point in time can anticipate a release of pages that will enable them to execute a lot faster than when the system is paging. At the same time, memory-resident jobs that have reached their steady-state can execute faster if they are not interfering with other memory-bound jobs and the kernel paging daemon does not reclaim pages from them. This scheme is friendly with short-lived sequential jobs that need fast response time. Another important aspect of the strategy is that it is not intrusive, since jobs claim no resources other than those granted to them by the operating system.

A detailed description of our dynamic paging prevention algorithm and its implementation in Linux is given in [13]. We provide a brief outline in Section 3 for the sake of clarity. The rest of the paper focuses on the integration of paging prevention and coscheduling.

3. Paging Prevention

The proposed paging prevention mechanism detects memory pressure in the same way that the operating system detects memory pressure before invoking the swapper, i.e. by checking if the allocated physical memory exceeds a predefined threshold. However, instead of evaluating the paging criterion inside the kernel, the evaluation is performed by the runtime system, at specific points of execution. The kernel exports the variables required to evaluate the paging criterion as symbols that can be accessed directly from shared memory.

Programs evaluate memory pressure against their memory demands. To obtain accurate estimates of the memory demands of programs and prevent paging, we intercept all memory allocation calls[2] and record the requested amount of memory in a hash table indexed by `pid` and 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, but it is coherent with the idea of preventing rather than reducing paging. The intention of the algorithm is to prevent paging by suspending the threads of jobs which are likely to incur paging. Estimation of the working set of a job requires that the job is executed until it reaches a steady state (the identification of which at runtime is a challenging problem by itself) and can be used to reduce the paging rate but not to prevent paging.

The algorithm is invoked from the programs and the kernel in an asymmetric manner. The programs invoke the algorithm at memory allocation points, where they can detect if the amount of memory requested by a program will reduce the amount of available physical memory below the kernel swapping threshold. The programs invoke the algorithm also at communication and synchronization points, immediately after the completion of the required communication operations. The rationale behind this is that completion of communication usually implies the start of a computation phase which may acquire memory.

The aforementioned invocation points can be statically identified in the runtime libraries linked to the programs, i.e., the C library for memory allocation points, and the communication library for communication points. Memory allocations done by the compiler are detected by the operating system, because our mechanism intercepts all `mmap` and `brk` calls to the kernel. To circumvent the problem of modifying system libraries, we are considering an alternative implementation based on dynamic instrumentation.

If a job detects that it is likely to incur paging, it suspends all its threads simultaneously and notifies the kernel scheduler to bypass them in future scheduling tournaments, until memory pressure is alleviated. These threads are resumed and compete again for CPU time as soon as the kernel detects that the system's memory can accommodate the suspended jobs. In that case, the kernel puts these jobs in the run queue at the first scheduling opportunity. Threads that were previously suspended due to memory pressure are serviced strictly on a FCFS basis, whenever they make attempts to acquire CPU time after suspension. Doing otherwise might force previously suspended jobs with large memory demands to starvation.

The way the algorithm is used by the programs raises some questions about its viability. More specifically, the algorithm appears to prevent paging only if all jobs running in the system cooperate and back-off after detecting memory pressure. One problem is what happens if the system runs greedy jobs that do not use our algorithm to adapt to memory pressure. We address this problem to a significant extent in the implementation of the algorithm, by intercepting the memory allocation requests made by all programs. The interception mechanism is applied universally (all `malloc` and `brk` calls are intercepted), therefore it is straightforward to capture jobs that attempt to allocate an amount of

[1] Our terminology assumes that each job is composed of one or more *processes*, potentially distributed across the cluster, and each process may fork one ore more threads of execution within a node of the cluster.

[2] Memory allocation requests to the operating system are issued with two system calls, `mmap` and `brk`.

memory that will force the kernel to reclaim pages. Based on the interceptions, we apply a simple scheduler extension that prevents the seemingly greedy jobs from allocating memory and continue with execution, by forcing memory allocation to fail and jobs to retry until sufficient memory is available to accommodate their data sets.

The anticipatory heuristic used in our algorithm presupposes that memory pressure is a short-lived event. This is true for the types of workloads we are considering, that is, parallel and distributed jobs interfering with unpredictable sequential load in each node. However, memory pressure may also be a long-term event, if the nodes of the cluster run memory-hungry simulations or similar memory-consuming jobs for periods of hours, days, or even weeks. This situation will most likely lead to starvation of jobs that back off after detecting memory pressure. Although we do not provide a solution for this case in this paper, we provide the interface to implement several alternatives. The most viable solution appears to be the rescheduling of the computation of the starving job, either by migrating its threads to nodes that have ample free memory, using algorithms like the ones proposed in [5, 18], or by redistributing the computation among threads that run on less loaded nodes.

4. Coscheduling and Paging Prevention

4.1. Implicit and Dynamic Coscheduling

The purpose of coscheduling is the coordinated allocation of processors to communicating processes. For parallel jobs in particular, coscheduling ensures that when the processes in the job need to communicate, they all have the CPU time needed to complete the communication. Implicit coscheduling is a decentralized coscheduling methodology, based on information available locally in each node. It constitutes a scalable alternative to explicit coscheduling [14], which attempts to control the allocation of processors to jobs using centralized or hierarchical control schemes.

Among several implicit coscheduling algorithms that appear in the literature, we selected a *periodic dynamic coscheduling* algorithm [11]. Dynamic coscheduling algorithms implement coscheduling by increasing the priority of processes that receive or send messages over the network. In our implementation, the operating system increases periodically the priority of processes, 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 the priority of which has to be increased. 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).

We selected the aforementioned dynamic coscheduling algorithm for two reasons. First, because coscheduling with dynamic priorities, as opposed to coscheduling with controlled waiting algorithms, seems to be the most efficient coscheduling technique for jobs with arbitrary or unknown communication patterns [11]. Second, because dynamic coscheduling 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 coscheduling in Linux required no more than 30 lines of C code, which were added to two files in the kernel source tree. On the contrary, a full-fledged implementation of implicit coscheduling with waiting algorithms requires non-trivial modifications to the communication libraries and the kernel, as well as microbenchmarks that obtain values for tunable parameters, such as round-trip communication times.

4.2. Combining Coscheduling and Paging Prevention

In theory, coscheduling and paging prevention operate along orthogonal axes. The goal of coscheduling is to ensure that two processes that need to communicate run at the same time on different nodes. The paging prevention algorithm on the other hand uses an anticipatory heuristic that tries to increase the throughput of a node by letting memory-resident jobs proceed without having their pages reclaimed by the operating system.

Coscheduling and paging 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, coscheduling is beneficial. In the symmetric case, a non-communicating process that does not fit in memory should be suspended to prevent paging. Non-communicating processes that fit in memory can be handled easily by the operating system. The more involved case is the one in which a communicating process does not fit in memory.

If both coscheduling and paging prevention are applied when a communicating process does not fit in memory, coscheduling takes priority over paging prevention, until the communicating process consumes the pending messages. Recall that our paging prevention algorithm is invoked after communication is completed. Whether this is the best choice or not, depends on the structure of communication and computation in the application, as well as memory utilization at the time when the process needs to communicate.

Assume that a process with a resident set that does not fit

in memory has not started paging yet. If this process reaches a communication point and still doesn't page, coscheduling will improve response time by helping the process —and implicitly any peers stalled at the communication point— to proceed with their computation. On the other hand, if a process is suspended by the paging prevention algorithm due to memory pressure, and shortly after that some messages destined to this process arrive at the network interface, coscheduling is in conflict with the paging prevention algorithm. This happens because coscheduling attempts to raise the priority of a process which is suspended to prevent paging. Whether this is a right decision or not depends on how the cost of paging compares to the earnings from coscheduling. If scheduling the process is enforced despite memory pressure, the process will make progress but will also pay the cost of paging, which lies on the critical path. 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.

Our scheduling algorithm strives for the latter option, i.e. giving priority to paging prevention. Although a formal analysis may be applicable in this case, our decision is driven by the results of our benchmarks (presented in Section 5). These results reveal that at high memory utilization levels, the penalty for paging exceeds by far the penalty for uncoordinated scheduling of communicating processes. In our implementation, giving priority to paging prevention over coscheduling is done as follows: the code that implements coscheduling in the kernel refrains from increasing the priority of processes marked as suspended due to memory pressure and these processes remain suspended in the kernel as long as memory pressure persists.

Coscheduling is implemented by periodically increasing the priorities of communicating processes, every time the kernel scheduler is invoked. Priority adjustments are computed using the following formula:

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

where b_t is the priority adjustment, L_t is the number of packets sent and/or received during period $(t-1, t)$, N 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 operating system. The heuristic uses the recent history of message traffic to/from the process. In our Linux implementation, the interval $(t-1, t)$ is set equal 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 running in the system, we modified the scheduler so that any extra time allocated to communicating processes due to priority boost is taken back by the kernel as soon as it detects that these processes have no more messages to send or consume. In this way, 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. Further details on the implementation can be found in [12].

5. Results

Previous evaluations of coscheduling algorithms used static workloads with a fixed number of memory-resident parallel jobs competing for resources. These types of workloads are convenient because they provide a good level of control over the execution environment and help the interpretation of results according to specific job characteristics such as communication volume, structure and frequency.

We follow a different path in our evaluation. We are assessing the impact of having a parallel job compete with random sequential load running in each node. In this scenario, we assume that each node of the cluster runs a typical server load, modelled with varying job demands in terms of CPU time and memory. We produced such a workload using heavy-tailed Pareto distributions for the lifetimes and the memory demands of jobs[3]. We set the mean CPU time of the jobs to 1 second and 60 seconds, to model both short-lived and long-lived server processes. The mean size of the memory footprint of the jobs is varied from 1 to 32 megabytes.

Although our initial experiments modelled also the interarrival time of jobs using a Poisson process, a study of workload traces from real academic servers [7] has shown that there is very low correlation between the interarrival times of jobs in a daytime workload. Therefore, in a second set of experiments, we opted for a workload that constantly feeds the nodes with jobs. Whenever a job completes its execution, another job with lifetime and memory demand drawn from the Pareto distribution takes its place. In this way, the workload sustains a roughly constant CPU utilization (the processor or processors in a node are kept busy all the time by sequential jobs of the workload) and the variation in throughput stems only from the memory demands of jobs, which place different levels of pressure on memory.

Two points are worth noting about the workload. First, although we don't explicitly control the amount of memory requested by the workload, we do enforce paging in two ways. The Pareto distribution ensures that there are always jobs that request large amounts of memory. In particular, we force the distribution to produce at least one job that requests the maximum amount of memory available in the node. Furthermore, increasing the size of the footprints of

[3]The rule of thumb in a heavy tailed distribution is that the probability that a process that already consumed more than 1 unit of a resource will consume more than T units, $T > 1$, is $\frac{1}{T^k}$ where $k \approx 1$.

the jobs implies an increase in the page fault rate. At some cross-over point, the system thrashes[4]. The second point worth noting is that the jobs in the workload do make use of our paging prevention algorithm. This means that local jobs with large memory footprints that run as part of the workload are cooperating with the scheduler to improve throughput. Therefore, the case of non-cooperating jobs discussed in Section 3 is non-existent.

The actual experiment consists of launching the workload and at the same time, launching back-to-back executions of a single compute-intensive parallel program. The duration of the experiment was arbitrarily set to 1 hour when the jobs in the workload have average lifetime equal to 1 second and 10 hours when the jobs in the workload have average lifetime equal to 60 seconds. We measure the average slowdown of the parallel program.The slowdown is obtained by dividing the execution time of the program while running with the workload in the background, to the execution time of the program while running alone on idle nodes.

For the purposes of the evaluation we selected one benchmark from the NAS benchmark suite, namely CG. In the NAS 2.3 distribution, CG is implemented with MPI. The benchmark computes the smallest eigenvalue of a sparse matrix using the Conjugate-Gradient method. The implementation uses an irregular communication pattern, which is a good test for coscheduling algorithms. We used the Class A problem, in which the size of the matrix is 14000 elements. The benchmark has a reasonably large data set that occupies approximately 64 megabytes of memory.

The experiments were conducted on a cluster with four 4-way Intel SMPs. Each SMP has Pentium Pro processors clocked at 200 Mhz, 512 kilobytes of external L2 cache per processor, and 1 gigabyte of memory. The sequential execution time of CG in this cluster is 63 seconds, while the parallel execution time on 4, 4-way nodes with Ethernet communication is 13 seconds. In the original experiments, the nodes ran version 2.4.2 of the Linux kernel. We conducted more experiments with a newer version of the kernel (2.4.16), because the implementation of the virtual memory system of Linux changed between releases of the 2.4 kernel and several of the problems of the original implementation were solved. The cluster uses regular TCP/IP communication over Fast Ethernet.

5.1. Results

Figure 1 shows the mean slowdown of CG, computed over back-to-back executions of the benchmark for one hour, during which the sequential workload ran on all four cluster nodes. The labels in the chart correspond to the unmodified Linux kernel (2.4.2), the kernel with the scheduler patched to enable dynamic coscheduling (2.4.2-cosch), the kernel extended with the paging prevention module (2.4.2-pp), the kernel with both the coscheduling patch and the paging prevention module, and paging prevention taking priority over coscheduling (2.4.2-pp/cosch) and the same setting with coscheduling taking priority over paging prevention (2.4.2-cosch/pp). Section 4 discussed the trade-off between these two options.

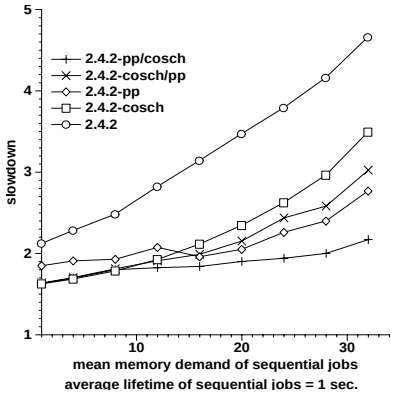

Figure 1. Slowdown of CG at various degrees of memory pressure.

As expected, increasing the memory demand of the workload increases the paging rate, which in turn slows down the parallel program. Linux has a noticeable sensitivity to paging, which is exacerbated even if the average memory demand of jobs is as low as 1 megabyte. This behavior is not to be expected and it has to do with anomalies in the page replacement algorithm of the Linux virtual memory system. Linux replaces blindly critical pages needed by running applications, as observed in [8].

At relatively low levels of memory pressure, when the average memory demand of the jobs in the workload ranges between 1 and 12 megabytes, coscheduling is the critical factor that improves the performance of the parallel program. Coscheduling appears to be more important than paging prevention, indicating that the impact of loosing coordination is larger than the impact of paging at low memory utilization levels. Note that the 2.4.2-cosch version uses the unmodified Linux VM system, whereas the 2.4.2-pp, 2.4.2-pp/cosch and 2.4.2-cosch/pp versions use our kernel module to prevent paging.

When the average memory demand increases beyond the threshold of 12 megabytes per job, the importance of paging prevention becomes clear. The kernel version that uses dynamic coscheduling but no paging prevention follows the diminishing returns of the unmodified kernel, resulting in a slowdown of more than 3 when the average size of the foot-

[4]Earlier work [13] indicates that the original Linux 2.4 kernel thrashes at surprisingly low levels of memory utilization. This result was also observed in [8]. However, newer versions of the kernel have a much more robust implementation of the VM system.

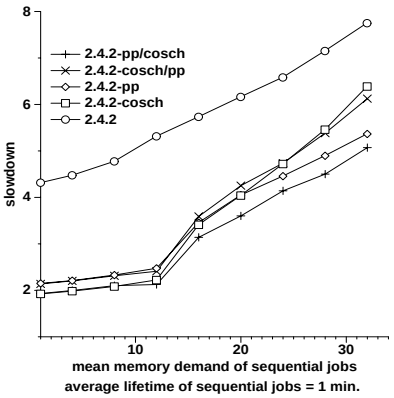

Figure 2. Slowdown of CG when competing with longer sequential jobs.

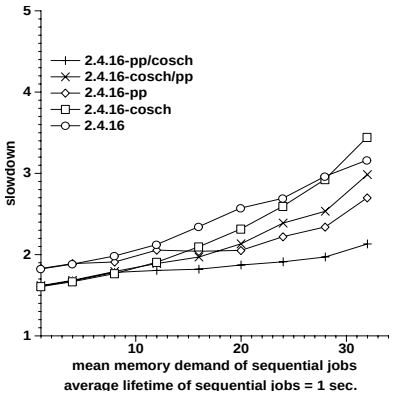

Figure 3. Results with a newer version of the Linux kernel (2.4.16).

print of sequential jobs reaches 32 megabytes. One interesting result is that paging prevention by itself reduces the slowdown of the non-coscheduled version of the program by more than 40%. An additional benefit of at least 20% is provided by coscheduling, but only if paging prevention takes priority over coscheduling. If coscheduling takes priority over paging prevention, performance is penalized by 25%. We mention here that the slowdown of the sequential jobs in the workload (not shown in the charts) is also reduced significantly (by a factor of 2.5 on average) when paging prevention is activated.

Figure 2 shows that if the sequential jobs have minute-long lifetimes, the slowdown of the parallel job is almost doubled. Although the relative trends of different scheduling choices do not change, it is clear that our scheduling module suffers from starvation. When the parallel job backs off due to memory pressure, it has to wait longer, because the resident jobs of the workload take longer to release memory. This type of problem requires more aggressive scheduling approaches, such as thread migration [7, 8, 18], or dynamic management of parallelism within the MPI job. We did not address this problem in the context of this work and this is a limitation that needs to be considered. What we view as positive is the fact that our framework provides the means for adaptation in the runtime system. The job can detect that it is starving and adapt using a variety of scheduling techniques. As such, our scheduling interface is orthogonal to the scheduling techniques needed to prevent starvation.

Figure 3 shows the improvements in the performance of the Linux VM system in a later release of the 2.4 kernel. Notice that there is absolutely no change in the relative performance of coscheduling and paging prevention, save an almost imperceptible improvement of less than 2%. The interesting result apart from the obvious improvement of the Linux VM system, is that coscheduling can actually hurt performance at high memory utilization levels. It appears that forcing coscheduling by increasing priorities when the average memory demand of jobs exceeds 28 megabytes, forces also the Linux kernel to suboptimal page replacements and reduces throughput.

6. Conclusions

This paper presented scheduler extensions for efficient execution of memory-bound parallel jobs on multiprogrammed clusters with limited memory resources. We have presented scheduling methods that combine coscheduling with adaptation to memory pressure. Adaptation is enabled by an operating system interface that informs the programs of memory utilization and gives them a chance to prevent paging via back-off. The same interface can be used to implement other more aggressive, and potentially more effective, adaptation schemes.

The paper has shown that neither coscheduling nor paging prevention is a panacea. These are two scheduling techniques that need to be combined for improving the throughput of clusters. Coscheduling is clearly required for tightly synchronized parallel programs and it is also a viable scheduling technique for client-server applications. Paging prevention might not be needed for lightly loaded clusters, but it is necessary for server settings. Further investigation is required in all directions (local scheduling, coscheduling, adaptability, and memory management) to understand the behavior of realistic workloads, if clusters are to be used as general-purpose computing farms rather than special-purpose compute servers.

Acknowledgement

We would like to thank the conference referees for many insightful comments that helped us improve the paper.

References

[1] A. Acharya and S. Setia. Availability and Utility of Idle Memory in Workstation Clusters. In *Proc. of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*, pages 35–46, Atlanta, GA, May 1999.

[2] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.

[3] A. Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Transactions on Computer Systems*, 19(3):283–331, Aug. 2001.

[4] A. Batat and D. Feitelson. Gang Scheduling with Memory Considerations. In *Proc. of the 14th IEEE International Parallel and Distributed Processing Symposium (IPDPS'2000)*, pages 109–114, Cancun, Mexico, May 2000.

[5] S. Chen, L. Xiao, and X. Zhang. Dynamic Load Sharing with Unknown Memory Demands in Clusters. In *Proc. of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, pages 109–118, Mesa, AZ, Apr. 2001.

[6] F. Giné, F. Solsona, P. Hernandez, and E. Luque. Coscheduling under Memory Constraints in a NOW Environment. In *Proc. of the 7th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'01), in conjunction with ACM SIGMETRICS'01*, Boston, MA, June 2001.

[7] M. Harchol-Balter and A. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, Aug. 1997.

[8] S. Jiand and X. Zhang. Adaptive Page Replacement to Protect Thrashing in Linux. In *Proc. of the 5th Annual Linux Showcase&Conference*, Oakland, CA, Nov. 2001.

[9] C. McCann and J. Zahorjan. Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers. In *Proc. of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'95)*, pages 208–219, Ottawa, Canada, May 1995.

[10] R. Mills, A. Stathopoulos, and E. Smirni. Algorithmic Modifications to the Jacobi-Davidson Parallel Eigensolver to Dynamically Balance External CPU and Memory Load. In *Proc. of the 15th ACM International Conference on Supercomputing (ICS'2001)*, pages 454–463, Sorrento, Italy, June 2001.

[11] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. Das. A Closer Look at Coscheduling Approaches for a Network of Workstations. In *Proc. of the 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 96–105, St. Malo, France, June 1999.

[12] D. Nikolopoulos and C. Polychronopoulos. Adaptive Scheduling under Memory Pressure on Multiprogrammed Clusters. Technical report, CSRD, University of Illinois at Urbana-Champaign, Dec. 2001.

[13] D. Nikolopoulos and C. Polychronopoulos. Adaptive Scheduling under Memory Pressure on Multiprogrammed SMPs. In *Proc. of the 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS'2002)*, Fort Lauderdale, FL, Apr. 2002.

[14] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. of the 3rd International Conference on Distributed Computing Systems (ICDCS'82)*, pages 22–30, Miami, FL, Oct. 1982.

[15] E. Parsons and K. Sevcik. Coordinated Allocation of Memory and Processors in Multipocessors. In *Proc. of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'96)*, pages 57–67, Philadelphia, PA, May 1996.

[16] S. Setia. The Interaction between Memory Allocation and Adaptive Partitioning in Message-Passing Multicomputers. In *Proc. of the 1st Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'95), in conjunction with IEEE IPPS'95, LNCS Vol. 949*, pages 146–165, Santa Barbara, CA, Apr. 1995.

[17] I. Venetis, D. Nikolopoulos, and T. Papatheodorou. A Transparent Operating System Infrastructure for Embedding Adaptability to Thread-Based Programming Models. In *Proc. of the 7th European Conference on Parallel Processing (Europar'01)*, pages 504–513, Manchester, UK, Aug. 2001.

[18] X. Zhang, Y. Qu, and L. Xiao. Improving Distributed Workload by Sharing both CPU and Memory Resources. In *Proc. of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS'00)*, pages 233–241, Taipei, Taiwan, Apr. 2000.