Adaptive Scheduling under Memory Pressure on Multiprogrammed SMPs

Nikolopoulos, D., & Polychronopoulos, C. D. (2002). Adaptive Scheduling under Memory Pressure on Multiprogrammed SMPs. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)* (pp. 1-6). Institute of Electrical and Electronics Engineers Inc..
<https://doi.org/10.1109/IPDPS.2002.1015481>

Published in:

Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)

Queen's University Belfast - Research Portal:

[Link to publication record in Queen's University Belfast Research Portal](#)

General rights

Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy

The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Adaptive Scheduling under Memory Pressure on Multiprogrammed SMPs*

Dimitrios S. Nikolopoulos and Constantine D. Polychronopoulos
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main St. Urbana, IL 61801, U.S.A.
dsn@csrd.uiuc.edu, cdp@csrd.uiuc.edu

Abstract

We present a simple scheduling strategy that copes with the adverse effects of paging on multiprogrammed SMPs. We consider open, multiuser SMP servers, typically found in academic or industrial environments. Our strategy incorporates four uniquely combined features. It is adaptive, in the sense that the programs themselves take scheduling actions upon detecting memory pressure; it is dynamic, since programs detect the likelihood of paging at runtime by communicating with the operating system through a lightweight interface; it is preventive, because it takes scheduling actions before paging occurs; and it is non-intrusive, because the local scheduling actions taken by a program do not affect adversely, but act to the benefit of other programs sharing the system. We present an efficient implementation of our strategy in Linux and show with a realistic production workload that it can improve the response time of the Linux kernel under memory pressure by up to a factor of eight and the throughput by up to a factor of four.

1. Introduction

Symmetric multiprocessors are consistently gaining popularity, as the platforms of choice for enterprise servers and as building blocks for cost-effective supercomputers. It is well known to the parallel processing community that one of the factors that jeopardizes the performance of SMPs is *multiprogramming*, i.e. the sharing of the resources of an SMP among jobs with competing—and usually conflicting—resource requirements.

*This work is 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.

An important, yet not so deeply investigated, effect of multiprogramming is *memory pressure*. The problem is attributed to the limited capacity of the second level of the SMP memory hierarchy (i.e. the DRAM) and occurs whenever the resident sets of simultaneously executing programs overwhelm the available physical memory. The performance penalty of memory pressure can be severe, because the operating system might be forced to page data to and from the hard disk. Paging has a disproportionately large cost compared to the cost of accessing memory, while the slowdown of a job that experiences paging during its execution is totally unpredictable.

In this paper, we are concerned with the adverse effects of paging on the performance of jobs running on multiprogrammed SMPs. More specifically, we consider the problem in the case of open systems, in which a running job may sense the effect paging at any time during its execution, due to unpredictable fluctuations of the system load.

Previous work in the area considered the problem of memory pressure as a problem of admission control and devised scheduling strategies that permit or forbid the execution of jobs, based on a snapshot of memory occupancy at job submission time and an a-priori estimation of the memory requirements of each job.

We consider a more realistic scenario for open SMPs, in which jobs are submitted from multiple users in an uncoordinated fashion and without a-priori knowledge of the required resources, thus placing the burden of resource management entirely on the operating system. In these cases, it is desirable to utilize the idle cycles left from the local owners by executing jobs submitted remotely from other users. The dynamics of the workload of open SMP servers may provide several opportunities for scheduling optimizations, whereas strict admission control may unnecessarily underutilize the system due to poor estimates of the resource requirements of jobs. Although our work targets tightly coupled SMPs, it is expected to have a significant impact

on large-scale multiprogrammed parallel systems that use SMPs as their building blocks.

We present a simple scheduling strategy that attempts to prevent paging. The idea is to provide feedback from the operating system to the programs, so that the programs themselves prevent paging by voluntarily suspending their threads at specific execution points. Our strategy incorporates four uniquely combined features. First, it is *adaptive*, in the sense that the programs—rather than the operating system—take scheduling actions upon detecting memory pressure. Second, it is dynamic, because programs detect the likelihood of paging at runtime and take appropriate scheduling actions accordingly. This is enabled by communicating information about memory utilization to the programs, through a lightweight interface with the operating system. Third, the scheduling strategy is *preventive*. It takes scheduling action before paging occurs. Fourth, the scheduling strategy is *non-intrusive*, because the local scheduling actions taken by programs to prevent paging do not affect adversely but act to the benefit of other programs sharing the system.

We have implemented our scheduling strategy as an external module of the Linux 2.4 kernel, coupled with a runtime library that exports the required scheduling interfaces to the programs. We ran experiments using a realistic workload, where the job CPU and memory requirements are derived from *heavy-tailed* distributions. Our measurements indicate that under high memory pressure, our scheduling module improves the response time of the Linux kernel by up to a factor of eight and the throughput of the Linux kernel by up to a factor of four, without compromising fairness.

The rest of this paper is organized as follows. Section 2 discusses some related work and contrasts it with our approach. Section 3 gives details on our scheduling strategy. Section 4 outlines the experimental setting and Section 5 presents the results. Section 6 concludes the paper.

2. Related Work

Burger et.al. [2] and Setia [8] were among the first to consider the impact of paging on the performance of multiprogrammed multiprocessors. Both studies concluded that paging has a detrimental effect on parallel jobs. Burger et.al. showed that in the presence of paging, the best choice for the threads of parallel jobs is to busy-wait until paging is completed, rather than yielding their processors with local or universal context switches¹. Setia has shown that even with optimistic estimations for the cost of paging, paging should be avoided in its entirety and a job the resident set of which does not fit in memory should not be admitted for execution, until memory pressure is alleviated.

¹A universal context switch suspends all threads of a parallel job simultaneously.

A number of related papers proposed scheduling algorithms that try to avoid paging on multiprogrammed distributed memory multiprocessors using admission control (e.g. [1, 4]). Driven by observations similar to that of Setia, the vast majority of these works proposed processor allocation policies that admit jobs for execution only if their resident sets fit in memory. This is a major difference to our work, which considers scheduling in situations where paging may occur while a job is executing. Our scheduler extensions are orthogonal to the processor allocation policy of the operating system and can be easily integrated with different scheduling strategies, such as gang scheduling and dynamic space sharing.

Mills et.al. [5] proposed a set of algorithmic modifications to a specific parallel application (the Jacobi-Davidson eigensolver), to cope with the effects of multiprogramming on clusters of workstations. This work bears the strongest similarity to ours, since it relies on polling resource utilization variables maintained in the operating system and runtime application adaptability to improve scheduling. The main difference to our work is that Mills et.al. propose application-specific scheduling strategies, while we propose an application-independent strategy, which can be embedded in standard runtime libraries linked to the programs. Another important difference is that Mills et.al. considered a simple workload with one parallel and one competing sequential application, while we consider production workloads with random arrivals and departures of jobs.

3. Scheduling Strategy

Our scheduling strategy relies on two operating system services: dynamic detection of memory pressure and dynamic estimation of the memory requirements of jobs. The idea is to temporarily suspend the threads of jobs which are likely to force paging due to their memory requirements. The novelty of the scheduling strategy is that the programs themselves detect the likelihood of paging at runtime and suspend their threads voluntarily, in a non-intrusive manner. A treatment of this problem in the kernel scheduler would require a great deal of sophistication and the associated cost for estimating working sets and prioritizing jobs according to their expected paging rates.

Reducing the system overhead by providing additional scheduling flexibility to the user level is one key aspect of our strategy. Performance is the second key aspect. Intuitively, the strategy is expected to improve response time, because in a system with a dynamic workload, jobs with memory requirements that stress the capacity of the system at a specific point in time can anticipate a release of memory resources that will enable them to execute a lot faster than when the system is paging. On the other hand, memory-resident jobs that have reached their steady-state

can proceed faster if they are not interfering with memory-hungry jobs and the kernel paging daemon. A third aspect of our scheduling strategy is that jobs claim no additional resources other than those granted to them by the operating system. This property renders the strategy non-intrusive. We outline the main components of our scheduling strategy in the following paragraphs.

3.1. Detecting Memory Pressure

Operating systems use always a safety threshold on the amount of physical memory that can be allocated to user jobs. When the amount of allocated memory exceeds this threshold, the operating system invokes a swapping daemon which writes pages to disk. Pages are swapped according to a priority scheme based usually on some variant of the least-recently-used (LRU) algorithm.

Our scheduling strategy uses the same threshold to assess the likelihood of paging. However, instead of evaluating the paging criterion inside the kernel, the evaluation is done from the programs at specific points of execution. The kernel exports the variables required to evaluate the paging criterion as symbols that can be accessed directly from user programs. Although the natural way to access these symbols is through system calls, we use a simpler and significantly more efficient implementation, which makes the symbols accessible to programs directly from shared memory.

The kernel allocates a page in physical memory and maps it to a file in the `/proc` filesystem. The contents of this page are defined by a kernel-level data structure that includes the exported variables. User programs map a page of their virtual address space to the same file and allocate a data structure which is identical, in terms of types and size, to the kernel data structure that stores the exported kernel variables. In this way, the programs read the kernel variables with regular load instructions from the common memory-mapped file, without system calls or context switches. Earlier work by Polychronopoulos et.al. [7] coined the term *shared arena* for operating system interfaces implemented through common mappings to shared memory.

3.2. Estimating Memory Requirements

Our idea for estimating the dynamic memory requirements of jobs is to intercept system calls that perform requests for memory allocation on behalf of the jobs. In UNIX-like operating systems, these calls are *brk* and *mmap*. The interception mechanism simply records the amount of memory requested by each job and invokes the native system call with the original arguments.

User-Level (invoked upon memory allocations, thread spawns, and entries of parallel constructs)

- (1) evaluate paging criterion;
- (2) while (criterion evaluates to true) {
- (3) check the bit mask with the scheduling status of threads in the shared arena;
- (4) $\forall$ thread in status *running* {
- (5) atomically set thread status to *yield*;
- (6) yield the processor;}}

Kernel-Level (invoked on the return path from system calls and upon timer expirations)

- (1) poll the bit mask with the scheduling status of threads in the shared arena;
- (2) evaluate paging criterion;
- (3) if criterion evaluates to false {
- (4) $\forall$ thread in status *yield* {
- (5) atomically set the status of the thread to *running*;
- (6) resume the thread;}}

Figure 1. Scheduling algorithm.

Each job polls its memory requirements and evaluates them against the paging criterion, through the same shared memory interface used to export the kernel variables that track memory utilization. A possible source of inaccuracy with this technique is that the memory allocation requests of a job may not reflect accurately the size of the working set of the job, i.e. the number of pages that need to be resident in memory when the job runs in steady state. We decide to tolerate this inaccuracy, because we attempt to prevent paging. Estimation of the working set of a job would probably require to actually execute the job until it reaches a steady state (the identification of which at runtime is itself a challenging problem), thus running the risk of forcing paging for the sake of measuring the working set. We prefer to intercept memory allocation requests so that we can predict the size of the resident set of a job, instead of measuring it while the job is running². More informed algorithms may be applicable in this case and this is an issue for further investigation.

3.3. Scheduling under Memory Pressure

The scheduling algorithm that we use to prevent paging is shown in Figure 1. The algorithm is invoked from the programs and the kernel in an asymmetric manner. The programs invoke the algorithm at memory allocation points, upon every thread spawn and at the entry points of parallel execution constructs. The algorithm is executed on each

²Note that the kernel maintains the size of the resident set of each process in the process descriptor.

thread, before the thread proceeds with the execution of its part of the computation. The rationale behind these invocation points is that paging should be prevented before the programs proceed with the execution of the bulk of the program's computation, which might stress the memory system by claiming pages on a demand basis. The kernel invokes the algorithm on the return from system calls and upon expirations of the scheduler's time quantum, right before selecting the next process to run.

When the algorithm is invoked, either at user level, or at kernel level, it polls the memory utilization variables maintained in the shared arena (Section 3.1), to evaluate whether the system will start paging if execution proceeds. If the criterion evaluates to true, all threads of the program are marked to yield their processors. This is done by maintaining a bit mask with the scheduling status of each thread in the shared arena. Each bit denotes a status of *running* (0) or *yield* (1). The critical observation for this step of the algorithm is that all threads of the same process yield their processors at roughly the same time and that the change of the scheduling status of threads is performed atomically by the first thread to detect memory pressure. Exporting scheduling status information to the user level helps the jobs react faster and prevent paging.

Each thread is blocked at the point at which the paging criterion is evaluated, as long as memory pressure persists. The first time the kernel detects that memory pressure is alleviated, it changes the scheduling status of all threads marked as preempted to *running* in the shared arena and lets them compete for the CPU. This guarantees that the kernel will grant CPU time to these threads the soonest possible. Note that although programs may suspend their threads independently, the kernel attempts to resume all the suspended threads at once. In all cases, the threads of a program do not make any attempt to explicitly increase their scheduling priority, or set the priority of other threads. This makes the algorithm non-intrusive.

We implemented the kernel side of the algorithm as an external kernel module of Linux. The user-level calls for polling the shared arena and probing the kernel module are currently implemented in a runtime system, which is linked to programs as a dynamic shared object. Details on the implementation are available in [6].

4. Experimental Setting

Motivated by the work of Harchol-Balter and Downey [3], we assembled a representative workload for a typical academic SMP server running in production mode. The system has random arrivals of jobs. The CPU time and memory footprint of each job are drawn from *heavy-tailed* distributions. The most important characteristic of a heavy-tailed distribution of a variable t is that the probability that $t > T$

is T^k , where k is a negative constant.

We used a workload in which the CPU time and memory requirements of jobs are drawn randomly from Pareto distributions. We fixed the mean requested CPU time³ of the jobs to 1, 10 and 100 seconds and varied the mean size of the memory footprint of the jobs from 1 to 64 Megabytes. The mean CPU times are representative for a typical academic server running mostly small (1 s.), mostly medium (10 s.), or mostly large (100 s.) jobs. The varying mean memory footprint models different operating conditions for the server. Assuming a server with 1 Gigabyte of memory (a quite realistic value for modern servers), the upper limit of the mean memory footprint (64 Megabytes) is equal to 6.25% of the available physical memory, enabling a theoretical maximum degree of multiprogramming equal to 16.

The arrival times of jobs follow a Poisson process. The workload includes a mix of sequential and multithreaded jobs. 50% of the jobs are sequential and 50% of the jobs are multithreaded. The number of threads used in each multithreaded job is drawn from a uniform distribution and never exceeds the number of processors in the system.

The multithreaded jobs in the workload follow the simple fork/join model of parallelism. Throughout their lifetime, the jobs fork a number of threads, each of which walks back and forth an array in memory and updates a set of elements assigned exclusively to that thread. After walking the array the threads join and the fork/join cycle is repeated, until an alarm signal ends the program upon expiration of the requested CPU time.

The workload consists of 10000 jobs. We executed it on a 4-processor Intel server, with Pentium Pro processors running at 200 MHz, 512 Kilobytes of L2 cache per processor and 1 Gigabyte of DRAM memory. As primary performance metrics, we use the mean slowdown and the slowdown variance. The slowdown of a job is simply the ratio of the job's wall-clock execution time to the job's requested CPU time. The latter is drawn from the Pareto distribution. Note that the mean slowdown is computed over the entire set of jobs, both sequential and multithreaded. Sequential jobs invoke our algorithm at memory allocation points. The slowdown variance is a metric that characterizes the fairness of the scheduler, in the sense that it indicates if there are *classes* of jobs with slowdowns that deviate significantly from the mean slowdown.

In order to assess the scheduler with real parallel applications as well, we conducted experiments in which we executed the workload together with instances of the NAS benchmarks, parallelized for shared-memory with OpenMP. More specifically, in the workload's script, we added a chain of back-to-back executions of one NAS benchmark at a

³The CPU time of a job is equal to or less than the wall-clock execution time of the job. The latter includes operating system effects, i.e. context switches, page fault handling etc.

Figure 2. Mean slowdown.

time. We measured the mean slowdown of the NAS benchmark with varying mean sizes of the memory footprints of the jobs in the workload. The OpenMP versions of the benchmarks were compiled to multithreaded code by the PGI OpenMP compiler. We used the class A problem size.

5. Results

The following results compare the performance of the unmodified Linux 2.4.2 kernel (labelled *kernel* in the charts), to the performance of the same kernel linked with our scheduling module to prevent paging (labelled *kernel+paging prevention* in the charts).

Figure 2 shows the mean slowdown of the jobs in the workload. The slowdown is plotted versus the mean size of the memory footprint of the jobs. Larger memory footprints imply higher memory pressure. It is clear that if memory pressure increases beyond a certain point (mean memory footprint equal to 20 Megabytes or larger), the Linux scheduler is unable to handle the workload. With the mean memory footprint at 64 Megabytes (which should enable a multiprogramming degree of 11, according to the Linux kernel thresholds), the mean slowdown is 109. Our scheduling module reduces the mean slowdown by as much as a factor of eight. For mean memory footprints up to 32 Megabytes, the slowdown is no more than four, a value which we consider as quite acceptable from a user's perspective. Similar trends are observed with medium and large jobs (in terms of CPU time), although the impact of memory pressure is less, since the cost of page faults can be amortized over longer periods of execution time.

Figure 3 gives another view of the performance of our scheduling module. The charts show the percentage of jobs that complete their execution before the system *thrashes* and the throughput of the system (in jobs/min., shown as annotations to the curves). We define the thrashing point as the point at which the system starts to abort the execution

Figure 3. Percentage of jobs completed and throughput (in jobs/min.).

of jobs due to insufficient memory resources. The throughput is computed somehow arbitrarily, as the number of jobs completed when the workload is executed for 10 minutes.

With paging prevention, thrashing does not occur with mean memory footprints up to 48 Megabytes, while even if the mean memory footprint is increased to 64 Megabytes, the system is able to complete 93% of the jobs submitted for execution. On the contrary, the Linux scheduler thrashes when the mean memory footprint exceeds 28 Megabytes. With a mean footprint of 64 Megabytes, Linux completes merely 9% of the jobs. Our scheduling module improves the throughput of the system under the modelled conditions by a factor of four (from 513 jobs/min. to 2004 jobs/min.). Notice that with our module, throughput is relatively immune to memory pressure. Even if the mean memory footprint increases to 64 Megabytes, the kernel is able to sustain 80% of its maximum throughput.

The variance of slowdown that we measured in the experiments follows roughly the same pattern as the mean slowdown itself. The variance with the Linux kernel increases linearly with the mean size of the memory footprint and varies from 0.5 to 10 for the memory footprints used in the experiments. The variance with our kernel module remains roughly constant and does not exceed 0.3. In general, we did not observe any particular pattern that pointed out unfair treatment of a specific class of jobs (e.g. jobs with higher/lower memory requirements or jobs with higher/lower CPU times). This result indicates that our scheduling module is not intrusive.

Figure 4 shows the mean slowdown of the NAS benchmarks, when executed simultaneously with the workload. Note that the memory size of the NAS benchmarks is fixed. The workload in this case serves as background noise that slows down the NAS benchmarks. The NAS benchmarks have long CPU times and can amortize potential performance losses due to paging over several time intervals, dur-

Figure 4. Mean slowdown of the NAS benchmarks.

Figure 5. Mean slowdown of BT and SP, when executed with our workload in a clustered setting.

ing which the system operates under low memory pressure. Nevertheless, the performance trends remain the same. The scheduling module reduces the slowdown of the benchmarks by a factor of three. In all cases it keeps the slowdown at acceptable (from a user's perspective) limits. The Linux kernel exhibits intolerable slowdowns for two benchmarks (SP and LU), but somewhat lower slowdowns for CG, MG and FT. SP, LU and BT happen to be the most memory-hungry benchmarks in the NAS benchmark suite. The results verify the intuition that high-end applications with large memory requirements and long execution times are likely to benefit more from a scheduling strategy that prevents paging.

6. Conclusions and Future Work

We have presented a simple scheduling strategy that copes with the adverse effects of paging on multiprogrammed SMPs. The strategy enables programs to react to memory shortage dynamically, at runtime. We have pre-

sented an efficient implementation of this strategy in Linux and demonstrated that it improves radically the performance of the Linux scheduler under memory pressure, in terms of both response time and throughput.

We are investigating the integration of our scheduling strategy with coscheduling schemes, since this would enable us to use our scheduling module to improve the throughput of clusters. Figure 5 shows the mean slowdown of versions of NAS BT and SP parallelized with a hybrid MPI/OpenMP model and executed on two 4-processor Linux boxes connected via a Myrinet switch. These preliminary results indicate that if our scheduling module is combined with dynamic coscheduling, the throughput of clusters under memory pressure can be considerably improved.

References

- [1] A. Batat and D. Feitelson. Gang Scheduling with Memory Considerations. In *Proc. of the 14th IEEE International Parallel and Distributed Processing Symposium*, pages 109–114, Cancun, Mexico, May 2000.
- [2] D. Burger, R. Hyder, B. Miller, and D. Wood. Paging Trade-offs in Distributed-Shared-Memory Multiprocessors. In *Proc. of IEEE/ACM Supercomputing'94: High Performance Networking and Computing Conference*, pages 590–599, Washington D.C., Nov. 1994.
- [3] 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.
- [4] 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*, pages 208–219, Ottawa, Canada, May 1995.
- [5] 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*, pages 454–463, Sorrento, Italy, June 2001.
- [6] D. Nikolopoulos and C. Polychronopoulos. Adaptive Scheduling under Memory Pressure on Multiprogrammed SMPs. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana Champaign, Jan. 2002.
- [7] C. Polychronopoulos, N. Bitar, and S. Kleiman. Nano-Threads: A User-Level Threads Architecture. Technical Report 1297, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Mar. 1993.
- [8] 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, LNCS Vol. 949*, pages 146–165, Santa Barbara, California, Apr. 1995.