Realistic Workload Scheduling Policies for Taming the Memory Bandwidth Bottleneck of SMPs

Christos D. Antonopoulos[1]*, Dimitrios S. Nikolopoulos[1], and Theodore S. Papatheodorou[2]

[1] Department of Computer Science, The College of William & Mary, 118 McGlothlin-Street Hall, Williamsburg, VA 23187-8795. U.S.A.
cda,dsn@cs.wm.edu,

[2] High Performance Information Systems Lab, Computer Engineering & Informatics Department, University of Patras, 26500 Patras, GREECE
tsp@hpclab.ceid.upatras.gr

Abstract. In this paper we reformulate the thread scheduling problem on multiprogrammed SMPs. Scheduling algorithms usually attempt to maximize performance of memory intensive applications by optimally exploiting the cache hierarchy. We present experimental results indicating that - contrary to the common belief - the extent of performance loss of memory-intensive, multiprogrammed workloads is disproportionate to the deterioration of cache performance caused by interference between threads. In previous work [1] we found that memory bandwidth saturation is often the actual bottleneck that determines the performance of multiprogrammed workloads. Therefore, we present and evaluate two realistic scheduling policies which treat memory bandwidth as a first-class resource. Their design methodology is general enough and can be applied to introduce bus bandwidth-awareness to conventional scheduling policies. Experimental results substantiate the advantages of our approach.

1 Introduction

Conventional schedulers for shared-memory multiprocessors are practically organized around the well-known UNIX multilevel priority queue mechanism, with limited extensions for support of multiprocessor execution. These schedulers try to achieve a balanced allocation of threads to processors. They also favor *cache affinity*, by preserving a long-term association between threads and the processors they are executed on.

This paper argues that for memory-intensive multiprogrammed workloads, it is often the consumed memory bandwidth and not necessarily the cache affinity that should be considered as a first-class citizen in the development of an effective multiprocessor kernel scheduler. Our recent work [1] indicated that bus

* This work has been partially carried out while the first author was with the High Performance Information Systems Lab, University of Patras, Greece

saturation can be harmful enough to nullify the benefits of parallelism. This paper shows that in certain memory-intensive workloads, memory bandwidth saturation is more harmful for performance that the loss of cache affinity.

We introduce two realistic workload schedulers, Bus Bandwidth-Aware Round Robin (B^2ARR) and Bus Bandwidth-Aware Dynamic Space Sharing (B^2ADSS), that effectively control bandwidth consumption. The methodologies used for their design are general and can be applied to introduce bus bandwidth consciousness to conventional, bandwidth-oblivious policies. The effectiveness of the new scheduling policies is evaluated using multiprogrammed workloads which consist of instances of NAS benchmarks [2].

Cache affinity scheduling is well studied in previous work [3–5]. Depending on the workload, it may provide substantial impact over a naive scheduling algorithm. This paper shows that for memory-intensive workloads, bandwidth consumption generally has a more significant impact than cache affinity, therefore it should be integrated as a criterion in multiprocessor schedulers. However, our policies do not prevent the use of cache affinity heuristics, since they are in general orthogonal to the criteria used for cache affinity preservation. Cache affinity heuristics usually control the association between processor and threads, whereas our policies focus on the optimal selection of coscheduled threads.

Symbiotic job scheduling [6, 7] proposes the use of event monitoring hardware to infer the interference between threads on shared execution resources and make informed online scheduling decisions based on this interference. It targets simultaneous multithreaded processors. Out policies target more conventional SMPs, however they could benefit multithreaded processors as well. They use a single additional metric (bus bandwidth consumption) as opposed to a complete array of microarchitectural events, in the case of symbiotic job scheduling. They are implemented with the efficiency of on-line monitoring as a principle, as opposed to symbiotic co-scheduling, which is a simulation-driven study.

Scheduling with runtime metrics such as the runtime speedup of parallel applications in multiprogrammed workloads has been investigated in related work [8, 9]. Our algorithms also use runtime metrics to improve scheduling decisions, however they focus on memory bandwidth consumption, a specific aspect of system performance, with a farther goal of maximizing system throughput.

This paper is organized as follows: Section 2 outlines the software and hardware configuration of our experimental platform. In Section 3 we present experimental results which indicate that cache affinity is not as important as bus bandwidth consumption for the performance of multiprogrammed SMPs. Section 4 introduces B^2ARR and B^2ADSS, two bus bandwidth-aware scheduling algorithms. In Section 5 we provide experimental evidence on the efficiency of the new algorithms. Finally, Section 6 concludes the paper.

2 Experimental Platform Configuration

For the purposes of our work we have used the NANOS compilation and execution environment on a system running Linux 2.4.25. The environment consists

of an OpenMP compiler, a run-time threads package and a CPU manager. The front-end of the environment is NanosCompiler [10], an OpenMP Fortran77 compiler which creates executables that can dynamically adapt the degree of their parallelism to the available processors.

We have developed a customized user-level CPU manager for testing kernel scheduling policies, without actual kernel hacking. Our CPU manager borrows several ideas from the NANOS CPU manager which we co-developed for cache-coherent NUMA multiprocessors [11], but uses a simplified internal structure and interface. The CPU manager communicates its scheduling decisions to the applications to allow them adapt to their execution environment. Moreover, it allows them to recover from inopportune thread preemptions by resuming preempted threads at the expense of executing threads of the same application.

The policies we introduce exploit performance related information available by all modern processors through performance monitoring counters. We have experimented on a dedicated, bus-based SMP system. The system is equipped with 4 hyperthreaded Intel Xeon MP processors, running at 1.4 GHz, with 256KB L2 cache each. We had to disable hyperthreading due to limitations in the concurrent performance monitoring of threads executing on the same processor. The system is also equipped with 1GB main memory. The practically attainable bandwidth of the bus which connects processors to main memory has been experimentally evaluated to be 1797 MB/sec.

Throughout our experiments we have used class W, OpenMP versions of benchmarks from the NAS 2.3 suite [2]. We did not use a higher class of NAS, such as class A or B, because the memory footprints of most benchmarks are large with respect to the physical memory of our system. In any case, the computational weight of applications is not as important as their computation / memory transfers ratio. We have also used three synthetic microbenchmarks: FLUSH, BBMA and nBBMA. Each time FLUSH is executed on a processor, it completely flushes the cache and then reuses data from it until getting suspended by the scheduler. BBMA is similar to FLUSH. The sole difference is that, beyond flushing the cache, BBMA continuously performs back-to-back accesses to the main memory. A single instance of BBMA can bring the system bus close to the limit of saturation. nBBMA, in turn, causes practically negligible interference to both the cache of the processors that execute it and the system bus.

3 Sensitivity of Workload Performance to Cache Affinity and Bus Saturation

In this section, we present experiments that quantify the impact of cache affinity and memory bandwidth-saturation on the performance of multiprogrammed workloads. The experiments have been executed using the CPU manager with a round-robin scheduling policy. The use of the CPU manager allows applications to adapt to the available processors and to minimize the adverse effects due to the non-coscheduled execution of their threads.

We executed 2 sets of experiments in order to evaluate the effect of multiprogramming on cache performance and the effect of cache affinity on workload performance. In the first set, each application is executed alone, using 4 threads. In the second set, we execute each application, which again requests 4 threads, together with 1, 2, 4 and 8 instances of the FLUSH microbenchmark. The slowdowns and the normalized L2 cache miss rates (CMR) with respect to the standalone execution are depicted in Figure 1.

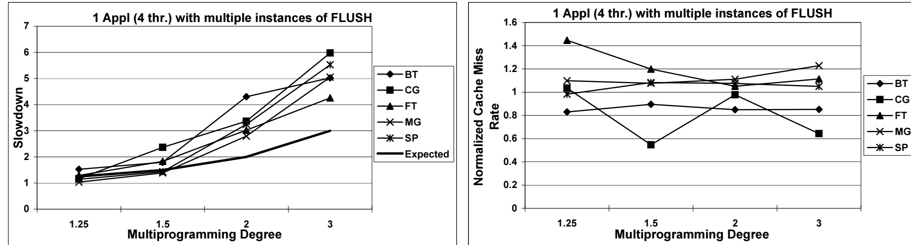

Fig. 1. Slowdowns (left) and normalized cache miss rates (right) from the multiprogrammed execution of NAS applications with instances of FLUSH.

Despite the close cooperation between applications and the CPU manager, applications suffer a performance penalty higher than expected, i.e. equal to the multiprogramming degree. Following the common belief, one would attribute the excessive performance penalty to the cache pollution introduced by FLUSH. However, the normalized CMR diagram reveals that, in most cases, the cache performance of this class of applications does not degrade severely due to FLUSH interference. Some workloads even experience a cache performance improvement in the presence of multiprogramming. As the CPU manager reduces the number of processors allocated to each application, applications react by reducing the number of threads they create and use. As a consequence, the effects of true- and false-data sharing are minimized. Moreover, in case two or more threads which share data happen to time-share the same processor, each one may benefit from the data fetched to the L2 cache by the others.

As a next step, we repeated the same experiments using nBBMA instead of FLUSH. nBBMA does not interfere with processor caches. If cache affinity was a determinative factor for workload performance, the slowdowns suffered by applications should this time be lower. The results are summarized in Figure 2. The comparison with Figure 1 reveals similar performance deterioration in both cases. The cache performance of applications is also similar (the diagram is not reported due to space limitations).

We then used BBMA in the workloads and repeated the experiments. BBMA continues causing traffic on the bus even after the cache has been flushed. Figure 3 depicts the slowdowns which are, this time, remarkably higher. The cache performance of applications is more diverse compared with the two previous

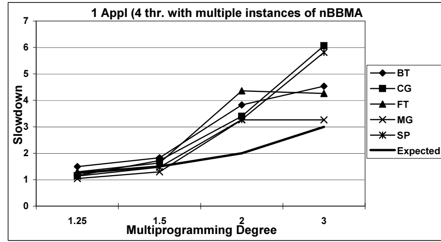

Fig. 2. Slowdown of the multiprogrammed execution of NAS applications with instances of nBBMA.

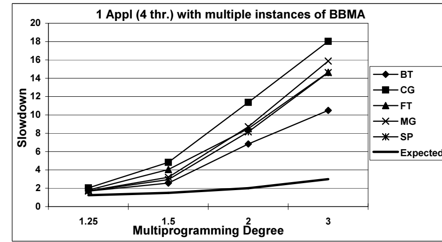

Fig. 3. Slowdown of the multiprogrammed execution of NAS applications with instances of BBMA.

experiments. However, there is again no clear cache performance deterioration trend which would justify the excessive performance loss.

These results are a strong indication that bus bandwidth is a valuable resource on bus-based SMP systems.The diverse effects of bus saturation prove to be more harmful for performance than the loss of locality on multiprogrammed, multiprocessor systems.

4 Bus Bandwidth Conscious Scheduling Policies

We introduce B^2ARR and B^2ADSS, two realistic scheduling policies which target bus bandwidth as a scheduling resource of primary importance. The policies are based on typical round-robin (RR) and on a variant of dynamic space sharing (DSS) presented in [12]. The scheduling quantum is fixed to 100 msec, equal to the scheduling quantum of the standard Linux scheduler. Ready to execute threads are conceptually organized as a linked list.

At the end of each scheduling quantum B^2ARR deallocates all executing threads are enqueues them to the tail of threads queue. It also updates the performance related statistics for all threads that executed during the latest quantum and calculates the Bus Transactions Rate (BTR) of each thread. BTR, measured as transactions/μsec, is a metric of the bandwidth consumption of the thread during its last execution and is used as an estimation of its future requirements. The Available Bus Transactions Rate (ABTR) is then initialized to be equal to the Systems Bus Transactions Rate (SBTR), a constant value which characterizes the system bus throughput. ABTR represents the available bus transactions rate for allocation to the remaining processors. It is calculated by subtracting the requirements of already allocated threads from SBTR.

Scheduling is divided in 2 phases. During the first phase, a portion of the system processors are allocated in a round robin way to the threads that reside at the head of the threads queue. Each time a processor is allocated to a thread, the BTR of that thread is subtracted from ABTR and the thread is dequeued from the threads queue. The remaining system processors are allocated to threads during the second scheduling phase, following bus bandwidth consumption cri-

teria. The processors are allocated in rounds, one processor at a time. At the beginning of each round the policy calculates the Average Bus Transactions Rate per Unallocated Processor (ABTR_{proc}). ABTR_{proc} corresponds to the bus transactions requirements of the ideal candidate for scheduling in this round. All threads in the queue are then scanned in order to locate the fittest thread for allocation in that round. Formula 1 estimates thread fitness:

$$Fitness = \frac{1000}{1 + |ABTR_{proc} - BTR|} \tag{1}$$

The fitness value quantifies the distance between the estimated (BTR) and the ideal (ABTR_{proc}) bus bandwidth consumption. At the end of each round a processor is allocated to the fittest thread. Formula 1 favors an optimal bus bandwidth exploitation. If, for example, threads with high bandwidth requirements have already been allocated, ABTR_{proc} is low and threads with low requirements are preferred for allocation. The formula works well even in cases bus saturation can not be avoided. If the bus gets overcommitted, ABTR_{proc} turns negative. As a result, the thread with the lowest bus bandwidth requirements is the fittest.

Bookkeeping and statistics collection are organized in B^2ADSS similarly to B^2ARR, however the scheduler also calculates the Average System Bus Transactions Rate (SBTR_{avg}), i.e. the average bus transactions rate of all active threads in the system. During the first phase of B^2ADSS , the typical DSS algorithm is applied to allocate processors to applications. However, it allocates a multiple *mult* of system processors ($mult * System\ Processors$). At the second phase, the scheduler forms *mult* chunks of threads to execute during the next *mult* quanta using bus bandwidth optimization criteria. Only threads that belong to applications which have been allocated processors during the first phase are candidates for selection. Formula 1 is used again for the fitness characterization of threads, however the target this time is to achieve a bus bandwidth utilization from each chunk of threads as close as possible to SBTR_{avg}.

The design of the new scheduling policies implicitly provides two general methodologies which can be applied to introduce bus bandwidth-consciousness to conventional, bandwidth-oblivious policies. The first method is to allocate a subset of system processors with the conventional policy. The remaining processors are allocated to threads with the goal of optimizing bus bandwidth consumption. The decision on the percentage of processors that are allocated by the conventional policy introduces an interesting tradeoff. If that percentage is low, threads with bus bandwidth requirements 'incompatible' with those of the other threads in the workload may experience large delays between two consecutive activations by the scheduler. On the other hand, allowing the conventional policy to allocate too many threads minimizes the opportunities for optimally co-scheduling threads that optimize bus bandwidth usage. In B^2ARR we have heuristically chosen to allocate 50% of the system processors using RR.

The second method applies the conventional policy to allocate processors to applications for a scheduling epoch, namely a number of scheduling quanta. The chunks of specific threads that execute during each quantum are then formed

with the objective of equilibrating bus bandwidth consumption among quanta. A similar tradeoff applies to this methodology as well. Using an epoch of few quanta may not allow an optimal exploitation of bus bandwidth. On the other hand, a wide epoch would introduce a significant delay between the first phase of scheduling and the actual execution of the threads in the last quanta of the epoch, increasing the risk of applying outdated scheduling decisions. For B^2ADSS the epoch has been heuristically chosen to be 2 quanta long.

The heuristic choice of the values of 50% and 2 for the percentage of processors allocated by the conventional policy and the epoch length respectively, has been experimentally driven. However, we intend to evaluate other heuristics as well. For example, it might be beneficial to allocate a percentage of the available bus bandwidth using the conventional policy, instead of a percentage of system processors.

In previous work [1] we have presented two variants of a scheduling policy which also schedules applications taking into account their bus bandwidth requirements. The two variants, namely Latest Quantum Gang (LQG) and Quanta Window Gang (QWG) are gang-like and target each application as a single entity. The fitness metric used to select and schedule applications is quite similar to Equation 1. LQG uses performance data collected only during the latest execution of each application, whereas QWG uses the average over a moving window which spans several previous quanta.

LQG and QWG, as typical gang-like policies, have the disadvantage of often resulting to suboptimal utilization of system processors. The rigid requirement of co-executing all application threads may leave processors idle during one or more scheduling quanta. B^2ARR and B^2ADSS alleviate this disadvantage by allowing an arbitrary number of threads of each application to execute simultaneously. On the other hand, the concurrent execution of all application threads minimizes synchronization and unbalancing problems which may appear due to inopportune preemptions of threads by the OS scheduler. However, the adaptability of applications created by the NanosCompiler, combined with the information and mechanisms offered by the CPU manager, allow threads scheduled with B^2ARR and B^2ADSS to minimize the adverse effects of such situations.

The new policies do not require any *a-priori* knowledge on the requirements of threads and their interaction with the hardware. Instead, they exploit performance data monitored in the past to predict thread behavior in the near future. This property, combined with the aforementioned characteristics make the new policies flexible and realistic. As a result, they are good candidates for adoption in a real-world system.

5 Experimental Evaluation

In order to evaluate the effectiveness of the new policies we have executed a set of workloads using the CPU manager with the new scheduling policies, the corresponding bandwidth-oblivious policies and LQG. Moreover, we have scheduled the same workloads using the native Linux scheduler, without the intervention

of the CPU manager. We did not experiment with QWG, since NAS applications have regular, smooth transaction patterns and are generally insensitive to external noise. For such applications, LQG performs better than QWG [1].

Table 1. Workload composition

ID	Description	Max. Multipr. Degree
A.1	2BT(3)+2CG(1)	2
A.2	4BT(1)+4CG(1)	2
A.3	2BT(4)+2CG(4)	4
B.1	2BT(1)+2SP(3)	2
B.2	4BT(1)+4SP(1)	2
B.3	2BT(4)+2SP(4)	4
C.1	(FT(1);FT(3))+(MG(3);MG(1))+ +(FT(3);FT(1))+(MG(1);MG(3))	3
C.2	4FT(1)+4MG(1)	2
C.3	2FT(4)+2MG(4)	4

Table 1 describes the workloads we have used. A(n) means that application A is executed with n threads. A+B represents the concurrent execution of applications A and B. Similarly, mA represents the concurrent execution of m instances of application A. Finally, A;B means that application B starts right after the termination of application A. The rightmost column of Table 1 reports the maximum multiprogramming degree exposed by each workload. However, the actual multiprogramming degree may vary during execution.

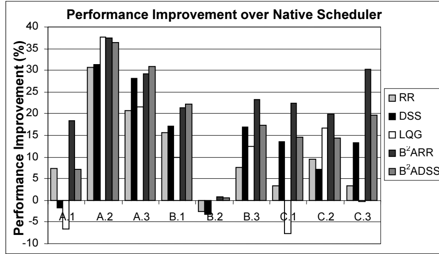

Fig. 4. Average workload turnaround time improvement, with respect to the execution with the native Linux scheduler.

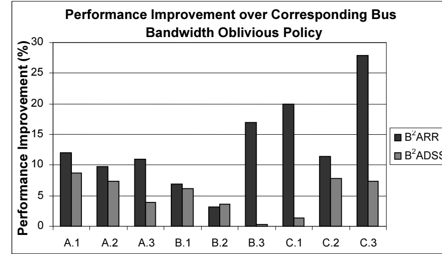

Fig. 5. Average workload turnaround time improvement when B^2ARR and B^2ADSS are applied, compared with the execution with RR and DSS respectively.

Figure 4 depicts the performance improvement attained by the new policies, executed in the context of the CPU manager, over the native Linux scheduler. A first important observation is that our CPU manager, even with bandwidth-oblivious policies such as RR or DSS, outperforms the native Linux scheduler by 11% and 14% respectively. This performance improvement can be attributed to the information and mechanisms provided to applications by the CPU manager, in order to assist them adapt to the available processors and make progress on

the critical path of their computation. Only in few cases (workloads A.1, B.2 and C.3) RR and DSS perform slightly worse (up to 3.2%) than the native scheduler. The average performance improvements attained by bus bandwidth-conscious policies, namely LQG, B^2ARR and B^2ADSS, are 9%, 23% and 18% respectively. In 2 workloads (A.1 and C.1) LQG performs worse than Linux scheduler. The reason is explained in detail in the next paragraph.

We then compare the performance of bandwidth-conscious policies with that of RR. The comparison isolates the performance gains due to application adaptability to the available processors and focuses on the impact of the policies themselves. LQG is in average 1% worse than RR. Although LQG performs in most cases better than RR, 3 workloads experience severe performance degradation. These workloads reveal the fundamental weakness of gang-like policies: the rigid rule of scheduling all threads of each application together results to low utilization of system processors. If the three problematic workloads are excluded, LQG is 4% more efficient than RR. In workload C.3 LQG also performs 4% worse than RR. In this case, RR forces the application to reduce the degree of parallelism in the presence of multiprogramming instead of time-sharing applications on the same processors. This choice proves to be beneficial for performance. B^2ARR and B^2ADSS, on the other hand, do not suffer from the same problems as LQG. They perform, in average, 28% and 17% better than RR.

As a next step, we quantify the performance improvement attained by B^2ARR and B^2ADSS over LQG. Although not co-scheduling application threads may introduce overheads, B^2ARR and B^2ADSS perform in average 13% and 9% better than LQG. Workloads A.2, B.2 and C.2 are the only exceptions. In these cases B^2ADSS performs up to 3% worse than LQG. Since all applications that participate in these workloads are single-threaded, they contribute equally to the total workload and the DSS phase of B^2ADSS has practically no effect.

Figure 5 summarizes the performance gains of bus bandwidth-aware scheduling policies over the corresponding bus bandwidth-oblivious ones. This comparison quantifies the performance improvement due to the optimal exploitation of bus bandwidth. It is important to notice that in all cases the new scheduling policies perform better than conventional policies. B^2ARR is in average 13% faster than RR. B^2ADSS also outperforms DSS, this time by 5%.

We expect our policies to perform even better on a system where more than 4 processors share a bus, since the bus saturation problem our policies cope with will be more acute on such a machine. However, commercial, bus-based SMPs are usually limited to 8 processors due to bus scalability issues. Multiple buses are used to integrate even 8 processors on a bus-based system. In such architectures, a slightly modified version of our policies would have additional choices for optimal bandwidth exploitation. It would be possible to even move threads among buses in order to optimize the bus bandwidth usage on each bus.

Significant effort has been paid to enhance the scalability of the Linux scheduler on large-scale multiprocessors. A new O(1) scheduler, present in 2.6.x kernels, allows constant overhead scheduling, independently of the number of tasks (N) and processors (P) in the system. The overhead related to the preservation of

load-balancing between processors grows linearly with the number of processors. The current implementation of our scheduling policies has an $O(N^2)$ overhead, which can be reduced to O(NlogN) if tasks are organized in priority queues, according to their BTR. The overhead is in any case higher than that of the standard Linux scheduler. However, as aforementioned, the policies target bus-based SMPs, which are limited to small- or medium-scales. For such systems the overhead of our policies has practically proven to be negligible.

6 Conclusions

In this paper we first presented experimental results which indicate that, for the class of memory-intensive, numerical applications, executed on multiprogrammed SMP systems, it is often bus bandwidth consumption and not cache affinity that determines application performance. Driven by this observation, we introduced B^2ARR and B^2ADSS, two realistic scheduling policies that target bus bandwidth as a top-importance scheduling resource. The new policies measure the bus-bandwidth requirements of threads during their execution and use the collected performance data to estimate thread behavior in the close future. They select threads to be co-scheduled during each quantum with the goal of neither wasting bus bandwidth, nor saturating the bus. The scheduling policies have been implemented in the context of a CPU manager, a user-level process that applies scheduling policies and precisely controls thread execution. The CPU manager is part of a compilation and execution environment which allows multithreaded applications to minimize the adverse effects of multiprogramming.

We evaluated the effectiveness of B^2ARR and B^2ADSS using workloads consisting of instances of applications from the NAS benchmarks suite. Our algorithms have been compared with the native Linux scheduler, the corresponding bus bandwidth-oblivious policies RR and DSS, and LQG, a gang-like, bus bandwidth-aware scheduling policy presented in our earlier work. B^2ARR and B^2ADSS attained significant performance gains over the Linux scheduler. Moreover, they turned out to be an important improvement over RR, DSS and LQG.

We plan to investigate the impact of sharing resources other than bus bandwidth on job scheduling. Such an investigation would be of particular interest for emerging architectures such as SMTs or HyperThreaded (HT) processors, where various execution units and levels of the on-chip memory hierarchy are shared among threads. We also intend to apply the idea of optimally using the available memory bandwidth to other levels of the memory hierarchy, beyond the front-side bus. Possible targets are the bandwidth of cache ports in SMTs, HT, and multi-core processors, or the bandwidth of network links in clusters of SMPs.

Acknowledgements

The first author is supported by a grant from 'Alexander S. Onassis' public benefit foundation and the European Commission through IST grant No. 2001-

33071. The first two authors are supported by NSF awards CARRER/CCF-0346867 and ITR/ACI-0312980.

References

1. Antonopoulos, C.D., Nikolopoulos, D.S., Papatheodorou, T.S.: Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. In: Proceedings of the 33d International Conference on Parallel Processing (ICPP '03), Kaohsiung, Taiwan, ROC (2003) 547–554
2. Jin, H., Frumkin, M., Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center (1999)
3. Squillante, M., Lazowska, E.: Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. IEEE Transactions on Parallel and Distributed Systems **4** (1993) 131–143
4. Torrellas, J., Tucker, A., Gupta, A.: Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. Journal of Parallel and Distributed Computing **24** (1995) 139–151
5. Vaswani, R., Zahorjan, J.: The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors. In: Proc. of the 13th ACM Symposium on Operating System Principles (SOSP'91), Pacific Grove, California (1991) 26–40
6. Snavely, A., Tullsen, D.: Symbiotic Job Scheduling for a Simultaneous Multithreading Processor. In: Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IX), Cambridge, Massachusetts (2000) 234–244
7. Snavely, A., Tullsen, D., Voelker, G.: Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In: Proc. of the ACM 2002 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'2002), Marina Del Rey, CA (2002) 66–76
8. Corbalan, J., Martorell, X., Labarta, J.: Performance Driven Processor Allocation. In: Proc. of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI'2000), San Diego, California (2000)
9. Nguyen, T., Vaswani, R., Zahorjan, J.: Maximizing Speedup through Self-Tuning Processor Allocation. In: Proc. of the 10th IEEE International Parallel Processing Symposium (IPPS'96), Honolulu, Hawaii (1996) 463–468
10. Ayguadé, E., Gonzàlez, M., Labarta, J., Martorell, X., Navarro, N., Oliver, J.: NanosCompiler: A Research Platform for OpenMP Extensions. Technical Report UPC-DAC-1999-39, Dept. D' Arquitectura de Computadors - Universitat Politècnica de Catalunya (1999)
11. Martorell, X., Corbalan, J., Nikolopoulos, D.S., Navarro, N., Polychronopoulos, E.D., Papatheodorou, T.S., Labarta, J.: A Tool to Schedule Parallel Applications on Multiprocessors. The NANOS CPU Manager. In: Proceedings of the 6th IEEE Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'2000). Volume 1911., LNCS (2000) 87–112
12. Polychronopoulos, E.D., Martorell, X., Nikolopoulos, D.S., Labarta, J., Papatheodorou, T.S., Navarro, N.: Kernel-Level Scheduling for the Nano-Threads Programming Model. In: Proceedings of the 12th ACM International Conference on Supercomputing (ICS'98), Melbourne, Australia, ACM Press (1998) 337–344