# Scheduling Algorithms with Bus Bandwidth Considerations for SMPs

Christos D. Antonopoulos[1*] Dimitrios S. Nikolopoulos[2†] Theodore S. Papatheodorou[1]
[1]High Performance Information Systems Lab
Computer Engineering & Informatics Dept.
University of Patras
26500 Patras, GREECE
cda, tsp@hpclab.ceid.upatras.gr

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

## Abstract

*The internal bus that connects processors to memory is known to be a major architectural bottleneck in SMPs. However, both software and scheduling policies for these systems generally focus on memory hierarchy optimizations and do not address the bus bandwidth limitations directly. In this paper, we first present experimental results which indicate that bus saturation can cause an up to almost three-fold slowdown to applications. Motivated by these results, we introduce two scheduling policies that take into account the bus bandwidth consumption of each application. The necessary information is provided by performance monitoring counters which are present in all modern processors. Our algorithms organize jobs so that processes with high-bandwidth and low-bandwidth demands are co-scheduled to improve bus bandwidth utilization without saturating the bus. We found that our scheduler is effective with applications of varying bandwidth requirements, from very low to close to the limit of saturation. We also tuned our scheduler for robustness in the presence of bursts of high bus bandwidth consumption from individual jobs. The new scheduling policies improve system throughput by up to 68% (26% in average) in comparison with the standard Linux scheduler.*

## 1 Introduction

Small symmetric multiprocessors have dominated the server market and the high-performance computing field, either as standalone components, or as components for building scalable clustered systems. Technology has driven the cost of SMPs down enough to make them affordable for desktop computing. Future trends indicate that symmetric multiprocessing within chips will be a viable option for computing in the embedded systems world as well.

This class of machines is praised for cost-effectiveness, but at the same time it is criticized for limited scalability. A major architectural bottleneck of SMPs is the internal bus which connects the processors and the peripherals to memory. Despite technological advances that drive the design of system-level interconnects to more scalable, switch-based solutions such as HyperTransport [4] and InfiniBand [5], the bandwidth of the internal interconnection network of SMPs is a dominant barrier for performance, especially when low-cost / low-performance buses are used.

Although it has been known for long that the internal bus of an SMP is a major performance bottleneck, software for SMPs has only taken indirect approaches to address the problem. The goal has always been to optimize the programs for the memory hierarchy and improve cache locality. The same philosophy is followed in SMP operating systems for scheduling multiprogrammed workloads with time-sharing. All SMP schedulers use cache affinity links for each thread. The affinity links bias the scheduler, so that each thread keeps running on the same processor. This helps threads build state in the caches without interference from other threads. Program optimizations for cache locality and cache affinity scheduling reduce the bus bandwidth consumed by programs. Therefore, they may improve the 'capacity' of the SMP in terms of the number of threads the SMP can run simultaneously without slowing them down. Unfortunately, if the bus of the SMP is saturated due to contention between threads, memory hierarchy optimizations and affinity scheduling do not remedy the problem.

In this paper, we present a direct approach for coping

*Supported by a grant from 'Alexander S. Onassis' public benefit foundation and the European Commission through the 'POP' IST project (grant No.: IST-2001-33071).

†Supported by a startup research grant from the College of William and Mary.

with the bus bandwidth bottleneck of SMPs in the operating system. We motivate this approach with experiments that show the impact of bus saturation on the performance of multiprogrammed SMPs. In our experiments we use applications with very diverse bus bandwidth requirements, which have already been extensively optimized for the target memory hierarchy. The experiments show clearly that this impact can be severe. The slowdown of jobs suffered due to bus bandwidth limitations can be significantly higher than the slowdown suffered due to interference between jobs on processor caches. In some cases, it is even higher than the slowdown the programs would experience if they were simply time-shared on the processor.

To address the problem directly, we propose scheduling algorithms which select the applications to run and assign processors driven by the bandwidth requirements of their threads. Bus utilization information is collected from the performance monitoring counters which are present in all modern processors. The algorithms measure the bandwidth consumption of each job at run-time. The goal is to find candidate threads for co-scheduling on multiple processors, so that the average bus bandwidth requirements per thread are as close as possible to the available bus bandwidth per unallocated processor. In other words, our policies try to achieve optimal utilization of the bus during each quantum without either overcommiting it or wasting bus bandwidth.

In order to evaluate the performance of our policies we experiment with heterogeneous workloads on multiprogrammed SMPs. The workloads consist of the applications of interest combined with two microbenchmarks: one that is bus bandwidth-consuming and another that poses negligible overhead on the system bus. The new scheduling policies demonstrate an up to 68% improvement of system throughput. In average, the throughput rises by 26%. A more detailed analysis of the work presented in this paper can be found in [3].

The rest of this paper is organized as follows: Section 2 discusses related work. In section 3 we present an experimental evaluation of the impact of bus bandwidth saturation on system performance. In section 4 we describe the new, bus bandwidth-aware scheduling policies. Section 5 presents an experimental evaluation of the proposed algorithms in comparison to the standard Linux scheduler. Finally, section 6 concludes the paper.

## 2 Related Work

Processor scheduling policies for SMPs have been primarily driven by the processor requirements and the cache behavior of programs. Most existing SMP schedulers use time-sharing with dynamic priorities and include an affinity mask or flag which biases the scheduler so that threads that have had enough time to build their state in the cache of one processor are consecutively scheduled repeatedly on the same processor. In these settings, parallel jobs can use all the processors of the system. Few SMP OSs use space sharing algorithms that partition the processors between programs so that each program runs on a fixed or variable subset of the system processors.

The effectiveness of cache affinity scheduling depends on a number of factors [10, 13, 15]. The cache size and replacement policy have an obvious impact. The smaller the size of the cache, the more the performance penalty for programs which are time-sharing the same processor. The degree of multiprogramming is also important. The higher the degree of multiprogramming, the less are the chances that affinity scheduling improves cache performance. The time quantum of the scheduler also affects significantly the effectiveness of affinity scheduling. With long time quanta, threads may not be able to reuse data from the caches. On the other hand, with short time quanta threads may not have enough time to build state on the caches.

Dynamic space sharing policies [8, 14] attempt to surpass the cache performance limitations by running parallel jobs on dedicated sets of processors, the size of which may vary at run-time. Thus, they tend to improve the cache performance of parallel jobs by achieving better locality. Their drawback is that they limit the degree of parallelism that the application can exploit. In most practical cases however, the locality improvement outweighs the loss of processors.

New scheduling algorithms based on the impact of cache sharing on the performance of co-scheduled jobs on multithreaded processors and chip-multiprocessors were proposed in [11, 12]. The common aspect of this work and ours is that both are using contention on a shared system resource as the driving factor for making informed scheduling decisions. However, these algorithms are based on analytical models of program behaviour on malleable caches, while our algorithms are using information collected from the program at run-time. Scheduling with on-line information overcomes the limitations of modelling program behaviour off-line, and makes the scheduling algorithm portable on real systems, regardless of workloads.

To the best of our knowledge, none of the previously proposed job scheduling algorithms for SMPs was driven by the effects of sharing system resources other than caches and processors. In particular, none of the policies was driven by the impact of sharing the bus, or in general, the network that connects processors and memory. Furthermore, among the policies that focus on optimizing memory performance, none considered the available bandwidth between different levels of the memory hierarchy as a factor for guiding the scheduling decisions.

Related work on job scheduling for multithreaded processors [1, 9] has shown that performance is improved when the scheduler takes into account the interference between

applications on shared hardware resources. More specifically, it is possible to achieve better performance on multiprogrammed workloads, if the programs co-scheduled on multiple processors during a quantum meet criteria that indicate good symbiosis on specific system resources. For example, the scheduler could co-schedule programs that achieve the least number of stall cycles on a shared execution unit. These works indicated the importance of sharing resources other than caches and processor time on the performance of job scheduling algorithms, but did not propose implementable scheduling algorithms driven by the observed utilization of specific resources.

Most modern microprocessors are equipped with performance monitoring counters. They provide the programmer with a powerful tool for tracking performance bottlenecks due to the interactions between the program and the hardware. These counters have been widely used for offline performance analysis of applications either autonomously [17] or as the basis for building higher-level tools such as Intel VTune Performance Analyzer. They have also been used as input to performance prediction functions [2], which can serve as prediction tools by extrapolating data collected from small, pilot executions. However, information attained from performance monitoring counters has never been used before at run-time to affect scheduling decisions on a real system, or drive program optimizations.

## 3 The Implications of Bus Bandwidth on Application Performance

In this section we present experimental results that motivate the investigation of new job scheduling policies which are driven by bus bandwidth consumption. Our results quantify the impact of sharing the bus of an SMP between multiple jobs. The experimental investigation is relevant for all types of shared-memory architectures that share some level of the memory hierarchy, that being a cache or RAM. Besides SMPs, the analysis is also relevant for multithreading processors and chip multi-processors.

For the experiments, we used extensively optimized applications and computational kernels from two suites, the NAS benchmarks [6] and the Splash-2 benchmarks [16]. The benchmarks have been compiled using the 7.1 version of Intel Fortran and C/C++ OpenMP compilers. We used codes which are hand-optimized for spatial and temporal cache locality in order to dismiss any chances that the observed bandwidth consumption occurs due to poor implementation of the used codes. We show that even with heavily optimized code, bus bandwidth consumption is a major limitation for achieving high performance.

Our experimental platform is a dedicated, 4-processor SMP with Hyperthreaded Intel Xeon processors, clocked at 1.4 GHz. It is equipped with 1 GB of main memory and each processor has 256 KB of L2 cache. The front-side bus of the machine (the bus connecting processors to memory) runs at 400 MHz. The operating system is Linux and the kernel version is 2.4.20. The hardware counters are monitored using Mikael Pettersson's performance counter driver for Linux and the associated run-time library. Unfortunately, the driver does not yet support concurrent execution of two threads on a physical processor if both threads use performance monitoring counters. As a consequence, we had to disable hyperthreading on all processors.

The theoretical peak bandwidth of the bus is 3.2 GB/s. However, the practically sustained bandwidth, as measured by the STREAM benchmark [7], is 1797 MB/s when requests are issued from all processors. The highest bus transactions rate sustained by STREAM is 29.5 transactions/usec. These measurements indicate that approximately 64 bytes are transferred with each bus transaction.

We have conducted 4 sets of experiments. The first one measures the bandwidth consumed by each application, when executed alone using 2 processors. The other three experiment sets simulate multiprogrammed execution. In the second set, two identical instances of an application are executed using 2 processors each.

In the third experiment set, one instance of the application using two processors runs together with two instances of a microbenchmark (BBMA). Each instance of the microbenchmark uses one processor. The microbenchmark accesses a two-dimensional array the size of which is twice the size of Xeon's L2 cache. The size of each line of the array is equal to the cache line size of Xeon. The microbenchmark performs column-wise writes on the array. More specifically, it writes the first element of all lines, then the second element and so on. The microbenchmark is programmed in C, so the array is stored in memory row-wise. Each write causes the processor to fetch a new cache line from memory. By the time the next element of each line is to be written, the specific line has been evicted from the cache. As a consequence, the microbenchmark has almost 0% cache hit rate. It constantly performs back-to-back memory accesses and consumes a significant fraction of the available bus bandwidth. In average, it performs 23.6 bus transactions/usec.

The fourth experiment set is identical to the third one, except from the configuration of the microbenchmark. The microbenchmark (nBBMA) accesses the array row-wise, so spatial locality is maximized. Furthermore, the size of the array is half the size of Xeon's L2 cache. Therefore, excluding compulsory misses, the elements are constantly accessed from the cache and the cache hit rate of the microbenchmark approaches 100%. Its average bus transaction rate is 0.0037 transactions/usec.

Figure 1A (black bars) depicts the bus bandwidth consumption of each application, measured as the number of

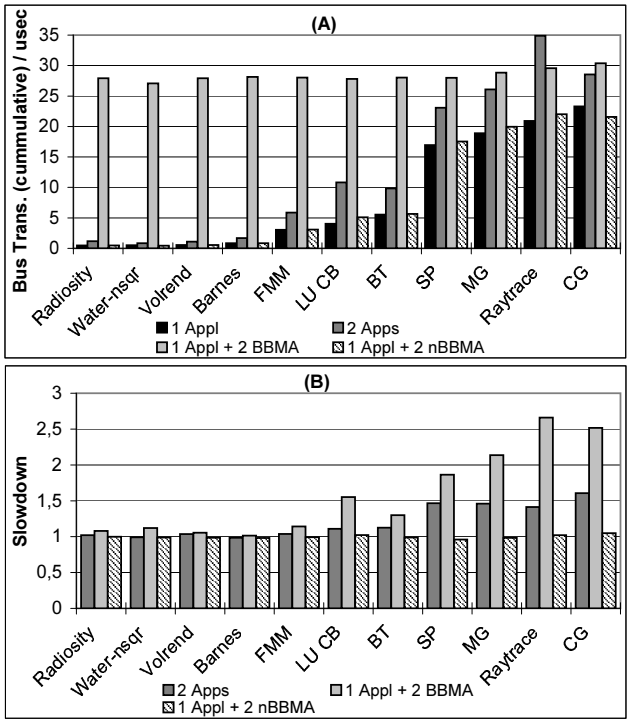


**Figure 1. Cumulative bus transactions rate (A) and slowdown (B) of applications when: i) applications are executed alone (black bars), ii) two instances of each application are executed simultaneously (dark gray bars), iii) one instance of each application is executed together with two instances of the BBMA microbenchmark (light gray bars) and iv) one instance of each application is executed together with two instances of the nBBMA microbenchmark (white, striped bars). Each application instance uses two processors.**

bus transactions per microsecond. The reported bus transaction rate is the accumulated rate of transactions issued from two threads running on two different processors. The applications are sorted in increasing order of issued bus transaction rate. The bandwidth consumption varies from 0.48 to 23.31 bus transactions per microsecond. Considering that each transaction transfers 64 bytes, the applications consume no more than 1422.73 MB/s, therefore the bus offers enough bandwidth to run these applications alone.

Figure 1A (dark gray bars) shows the accumulated number of transactions per microsecond, when two instances of each application run simultaneously using two processors each. Note that there is no processor sharing. The four applications with the highest bandwidth requirements (SP, MG, Raytrace, CG) push the system bus close to its capacity. Even in cases the cumulative bandwidth of two instances of these applications does not exceed the maximum sustained bus bandwidth, contention and arbitration contribute to bandwidth consumption and eventually bus saturation. Diagram 1B shows the corresponding slowdown applications suffer. The slowdown is calculated as the arithmetic mean of the slowdown of the two instances. Theoretically, the applications should not be slowed down at all, however in practice, there is slowdown due to contention between the applications on the bus. The results show that the applications with high bandwidth requirements suffer a 41% to 61% performance degradation. It is worth noting that four Raytrace threads yield a cumulative rate of 34.89 transactions/usec, which is higher than the transactions rate achieved by four concurrently executing threads of STREAM (29.5 transactions/usec). It has not been possible to reproduce this behavior with any other application or synthetic microbenchmark. We are currently investigating this issue in cooperation with Intel.

Figure 1 (light gray bars) illustrates the results from the experiments in which one parallel application competes with two copies of the BBMA microbenchmark which streams continuously data from memory without reusing them. These experiments isolate the impact of having applications run on an already saturated bus. Note that the bus bandwidth consumed from the workload is very close to the limit of saturation, averaging 28.34 transactions/usec. Memory-intensive applications suffer 2 to almost 3-fold slowdowns, despite the absence of any processor sharing. Even applications with moderate memory bandwidth requirements have slowdowns ranging between 2% and 55% (18% in average). The slowdown of LU CB is higher than expected. This can be attributed to the fact that LU CB has a particularly high cache hit ratio (99.53% when executed with two threads). As a consequence, as soon as a working set has been built in the cache, the application tends to be very sensitive to thread migrations among processors. The same observation holds true for Water-nsqr as well.

The white, striped bars correspond to the results from the concurrent execution of parallel applications - using two threads each - with two instances of the nBBMA microbenchmark. The latter practically poses no overhead on the bus. It is clear that both the bus transactions rate and the execution time of applications are almost identical to those observed during the uniprogrammed execution. This confirms that the slowdowns observed in the previously described experiments are not caused by lack of computational resources. These results also indicate that pairing high-bandwidth with low-bandwidth applications is a good way for the SMP scheduler to achieve higher throughput.

From the experimental data presented in this section, one can easily deduce that programs executing on an SMP may suffer significant performance degradation even if they are

offered enough CPU and memory resources to run without sharing processors and caches and without causing swapping. These performance problems can be attributed to bus saturation. In some cases, the slowdowns exceed the slowdowns that would have been observed if the threads were simply time-shared on a single processor, instead of executing on different processors of a multiprocessor. Given the magnitude of the slowdowns it is reasonable to search for scheduling policies that reduce performance penalties by carefully managing bus bandwidth.

# 4 Scheduling Policies for Preserving Bus Bandwidth

We have implemented two new scheduling policies that schedule jobs on an SMP system taking into account the bus bandwidth the jobs consume. They aim at optimizing the use of system bus bandwidth, by co-scheduling jobs that neither underutilize nor saturate the bus. The policies are referred to as '*Latest Quantum*' and '*Quanta Window*'. Both policies are gang scheduling-like. Processors are allocated to an application only if they are enough for all its threads to execute. The scheduling quantum is fixed to a constant value.

The applications controlled by our policies are conceptually organized as a list. At the end of each scheduling quantum, the '*Latest Quantum*' policy updates the bus bandwidth consumption statistics for all running jobs, using information provided by the applications. The bus bandwidth consumed per application thread ($\text{BBW}_{/thread}$) is calculated by equipartitioning the bandwidth requirements of each application during the latest quantum among its threads. The previously running jobs are then transferred to the end of the applications list.

Following, the policy elects the applications to execute during the next quantum. The application at the top of the applications list is allocated by default. This ensures that all applications will eventually have the chance to run, independent of their bus-bandwidth consumption characteristics. As a consequence, no job will suffer processor starvation.

Every time an application is elected to run, the available bus bandwidth in the system is calculated by subtracting the requirements of already allocated applications from the total bandwidth of the system bus. The available bus bandwidth per unallocated processor ($\text{ABBW}_{/proc}$) is then estimated as the remaining bandwidth divided by the number of unallocated processors.

As long as there are processors available, the scheduler traverses the list of applications. For each application that fits in the available processors, a fitness value is calculated.

$$Fitness = \frac{1000}{1 + |ABBW_{/proc} - BBW_{/thread}|} \quad (1)$$

Fitness is a metric of the proximity between the application's $\text{BBW}_{/thread}$ and the current $\text{ABBW}_{/proc}$. The closer $\text{BBW}_{/thread}$ is to $\text{ABBW}_{/proc}$ the fitter the application is for scheduling. The selection of this fitness metric favors an optimal exploitation of bus bandwidth. If processors have already been allocated to low-bandwidth applications, high-bandwidth ones become best candidates for the remaining processors. The reverse scenario holds true as well. The fitness metric behaves as expected even in cases when, due to the nature of the workload, bus saturation can not be avoided. As soon as the bus gets overloaded, $\text{ABBW}_{/proc}$ turns negative and the application with the lowest $\text{BBW}_{/thread}$ becomes the fittest.

At the end of each list traversal the fittest application is selected to execute during the next quantum. If there are still unallocated processors a new list traversal is performed.

'*Quanta Window*' policy is quite similar to the '*Latest Quantum*' one. The sole difference is that instead of taking into account the bus bandwidth requirements of each thread during the latest quantum, we use the average of its requirements during a window of previous samples ($\overline{BTR_{/thread}}$). Equation 1 can now be written as:

$$Fitness = \frac{1000}{1 + |ABTR_{/proc} - \overline{BTR_{/thread}}|} \quad (2)$$

Using $\overline{BTR_{/thread}}$ instead of $\text{BTR}_{/thread}$ has an effect of smoothing sudden changes to the bus transactions caused by an application. This technique filters out bursts with small duration or bursts that can be attributed to random, external events (for example when a thread migrates to another processor and rebuilds its state in the cache). However, at the same time it reduces the responsiveness of the scheduling policy to true changes in the bus bandwidth consumption. The selection of the window length must take this tradeoff into account. The window used in our experimental evaluation is 5 samples long. This length has the property of limiting the average distortion introduced by filtering within 5% of the observed transactions pattern for applications with irregular bus bandwidth requirements, such as Raytrace or LU. The use of a wider window would require techniques such as exponential reduction of the weight of older samples, in order to achieve an acceptable policy responsiveness.

In order to design and test our scheduling policies without altering the operating system kernel, we implemented a user-level CPU manager. The user-level CPU manager runs as a server process on the target system. Each application that wishes to use the new scheduling policies sends a 'connection' message to the CPU manager (through a standard UNIX-socket). The CPU manager responds to the connection message by creating a shared arena, i.e. a shared memory page which is used as its primary communication medium with the application. It also informs the

application how often the bus transaction rate information on the shared-arena is expected to be updated. In order to ensure the timeliness of information provided from the applications, the bus transaction rate is updated twice per scheduling quantum. At each sampling point the performance counters of all application threads are polled, their values are accumulated and the result is written to the shared arena. The CPU manager also adds a descriptor for each new application to a doubly linked circular list.

The applications are blocked / unblocked by the CPU manager according to the decisions of the active scheduling policy. Blocking / unblocking of applications is achieved using standard unix signals. The CPU manager sends a signal to an application thread which, in turn, is responsible to forward the signal to the rest of the application threads. In order to avoid side-effects from possible inversion in the order block / unblock signals are sent and received, a thread blocks only if the number of received block signals exceeds the corresponding number of unblock signals. Such an inversion is quite probable, especially if the time interval between consecutive blocks and unblocks is narrow.

A run-time library which accompanies the CPU manager offers all the necessary functionality for the cooperation between the CPU manager and applications. The modifications required to the source code of applications are limited to the addition of calls for connection and disconnection and to the interception of thread creation and destruction.

The overhead introduced by the CPU manager to the execution time of the applications it controls is usually negligible. In the worst case scenario, namely when multiple identical copies of applications with low bus bandwidth requirements are co-executed, it may rise up to 4.5%.

## 5 Experimental Results

In order to evaluate the effectiveness of our policies, we have experimented with three sets of heterogeneous workloads. Each set is executed either on top of the standard Linux scheduler, or with one of the proposed policies. All workloads have a multiprogramming degree equal to two. In other words, there are eight concurrently active threads, twice as many as the available physical processors. The scheduling quantum of the CPU manager is 200 msec, twice the quantum of the Linux scheduler. We have experimented with a CPU manager quantum of 100 msec, which resulted to an excessive number of context switches. This can be attributed to the lack of synchronization between the OS scheduler and the CPU manager, which results to conflicting scheduling decisions at the user- and kernel-level. Using a larger scheduling quantum eliminates this problem. In any case, we have verified that the duration of the CPU manager quantum does not have any measurable effect on the cache performance of the controlled applications.

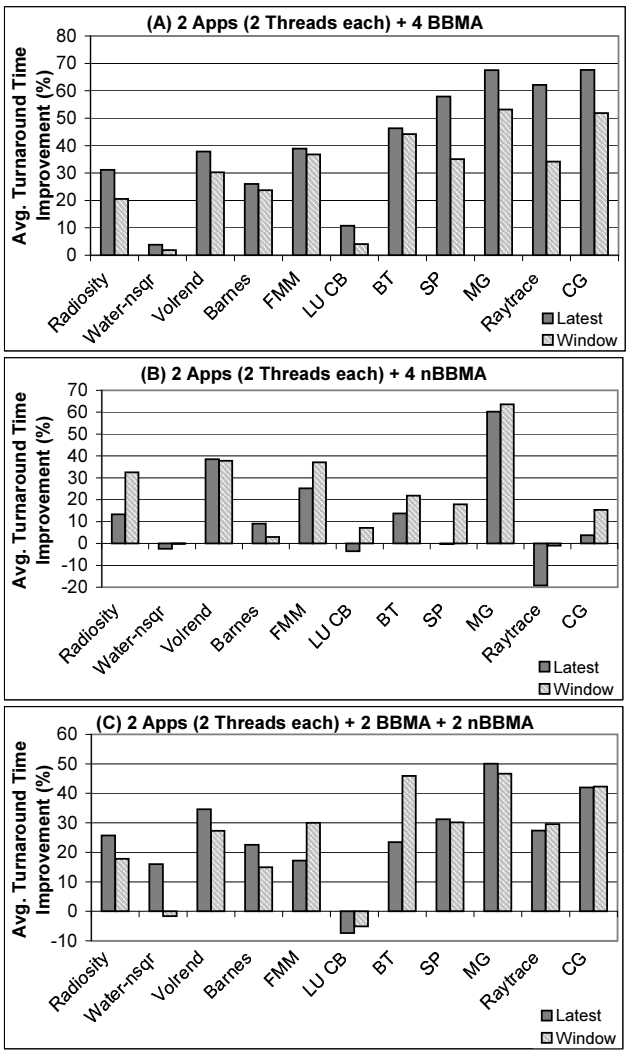


**Figure 2. Performance improvement (%) of the workloads when two instances of each application are executed simultaneously with i) four instances of the BBMA microbenchmark (A), ii) four instances of the nBBMA microbenchmark (B) and iii) two instances of the BBMA and two instances of the nBBMA microbenchmark (C). The reported values are the improvement in the arithmetic mean of the execution times of both application instances.**

In the first set, two instances of the target application, requesting two processors each, are executed together with four instances of the BBMA microbenchmark. This set evaluates the effectiveness of our policies on an already saturated bus. Figure 2A illustrates the improvement each policy introduces in the average turnaround time of the applica-

tions in comparison to the execution on top of the standard Linux scheduler. In all diagrams applications are sorted in increasing order of issued bus transactions rate in the uniprogrammed execution (as in figure 1A). The '*Latest Quantum*' policy achieves improvements ranging from 4% to 68% (41% in average). The improvements introduced by the '*Quanta Window*' policy vary between 2% and 53% with an average of 31%.

When executed with the standard Linux scheduler, applications with high bandwidth requirements may be co-scheduled with instances of the BBMA microbenchmarks, resulting to bus bandwidth starvation. Our policies avoid this scenario. Applications with lower bandwidth requirements may be scheduled with instances of the BBMA microbenchmarks. However, even in this case, our policies ensure - due to the gang-like scheduling - that at least two low-bandwidth threads will run together, in contrast to the Linux scheduler which may execute one low-bandwidth thread with three instances of BBMA.

The second set of workloads consists of two instances of the target application - requesting two processors each - and four instances of the nBBMA microbenchmark. This experiment demonstrates the functionality of the proposed policies when low bandwidth jobs are available in the system. Figure 2B depicts the performance gains attained by the new scheduling policies.

'*Latest Quantum*' achieves up to 60% higher performance, however three applications slow down. The most severe case is that of Raytrace (19% slowdown). A detailed analysis of Raytrace revealed a highly irregular bus transactions pattern. The sensitivity of '*Latest Quantum*' to sudden changes of bandwidth consumption has probably led to this problematic behavior. Moreover, from figure 1A one can deduce that running two threads of Raytrace together - which happens due to the gang-like nature of our policies - may alone drive the bus to saturation. LU CB and Water-nsqr also suffer minimal slowdowns due to their high sensitivity to thread migrations among processors. In average, '*Latest Quantum*' improved workload turnaround times by 13%. The '*Quanta Window*' policy turned out to be much more stable. It improved workload turnaround times by up to 64%. Raytrace slows down once again, this time by only 1%. The average performance improvement is now 21%.

In this experiment set, our scheduling policies tend to pair bandwidth consuming applications with instances of the nBBMA microbenchmark. As a consequence, the available bus bandwidth for demanding applications is higher. Even low-bandwidth applications seem to benefit from our algorithms. The new policies avoid executing 2 instances of the applications together in the presence of nBBMA microbenchmarks. Despite the fact that running two instances of low-bandwidth applications together does not saturate the bus, performance problems may occur due to contention among application threads for the possession of the bus.

The third experiment set combines two instances of the target application - requesting two processors each - with two instances of the BBMA and two instances of the nBBMA microbenchmark. Such workloads simulate execution environments where the applications of interest coexist with more and less bus bandwidth consuming ones. The improvements of the new scheduling policies over the Linux scheduler are depicted in figure 2C.

'*Latest Quantum*' policy improves the average turnaround time of applications in the workloads by up to 50%. LU is the only application that experiences a 7% performance deterioration. The average performance improvement is 26%. The maximum and average improvement achieved by '*Quanta Window*' are 47% and 25% respectively. Two applications, namely Water-nsqr and LU suffer minimal slowdowns of 2% and 5%.

In summary, for the purposes of this experimental evaluation we used applications with a variety of bus bandwidth demands. All experiment sets benefit significantly from the new scheduling policies. Both policies attain average performance gains of 26%. The scheduling algorithms are robust for both high- and low-bandwidth applications. As expected however, '*Quanta Window*' proves to be much more stable than '*Latest Quantum*'. It performs well even in cases the latter proves too sensitive to sudden, short-term changes in the bandwidth consumption of applications.

## 6 Conclusions

Symmetric multiprocessors are nowadays very popular in the area of high performance computing both as standalone systems and as building blocks for computational clusters. The main reason is that they offer a very competitive price/performance ratio. However the limited bandwidth of the bus that connects processors to memory has adverse effects to the scalability of SMPs. Although this problem is well-known, neither user- nor system-level software is optimized to minimize these effects.

In this paper we have presented experimental results which indicate that bus saturation is reflected to an almost 3-fold decrease in the performance of bus bandwidth consuming applications. Even less demanding applications suffer slowdowns between 2% and 55%.

Motivated by this observation, we introduced two scheduling policies that take into account the bus bandwidth requirements of applications. Both policies have been implemented in the context of a user-level CPU manager. The information required to drive policy decisions is provided by the performance monitoring counters present in all modern processors. To the best of our knowledge this is the first time these counters have been used to improve application performance at run-time. '*Latest Quantum*' policy

uses the bus transactions rate of applications during the latest quantum, whereas '*Quanta Window*' uses a moving window average. At any given scheduling point both policies try to schedule the application whose bus transaction rate per thread better matches the available bus transaction rate per unallocated processor in the system.

In order to evaluate the performance of our policies, we have executed three sets of workloads. In the first set, applications of interest coexisted with highly bus demanding microbenchmarks. The second set consisted of the applications of interest and microbenchmarks that pose no overhead on the bus. In the third set applications executed in an environment composed of both highly-demanding and not-demanding microbenchmarks. Both policies attained an average 26% performance improvement over the native Linux scheduler. Moreover, '*Quanta Window*' has been much more stable than '*Latest Quantum*'. It maintained good performance even in corner-cases where '*Latest Quantum*' proved to be oversensitive to application peculiarities.

We plan to continue this work in the following directions. First, we will derive analytic or empirical models of the effect of sharing resources such as the bus, caches and main memory, on the performance of multiprogrammed SMPs. Using these models, we can re-formulate the multiprocessor scheduling problem as a multi-parametric optimization problem and derive practical model-driven scheduling algorithms. We plan to test our scheduler with I/O and network-intensive workloads which stress the bus bandwidth, using scientific applications, web and database servers. Our work can also be extended in the context of multithreading processors, where sharing happens also at the level of internal processor resources, such as the functional units.

## References


[1] G. Alverson, S. Kahan, R. Corry, C. McCann, and B. Smith. Scheduling on the Tera MTA. In *Proc. of the first Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'95), LNCS Vol. 949*, pages 19–44, Santa Barbara, CA, Apr. 1995.

[2] N. M. Amato, J. Perdue, A. Pietracaprina, G. Pucci, and M. Mathis. Predicting Performance on SMPs. A Case Study: The SGI Power Challenge. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS 2000)*, Cancun, Mexico, May 2000.

[3] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. Technical Report HPCLAB-TR-090703, High Performance Information Systems Lab, University of Patras, July 2003.

[4] Meeting the I/O Bandwidth Challenge: How HyperTransport Technology Accelerates Performance in Key Applications. Technical report, HyperTransport Consortium, http://www.hypertransport.org/, December 2002.

[5] Infiniband Architecture Specification, Release 1.1. Technical report, Infiniband Trade Association, http://www.infinibandta.org, November 2002.

[6] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[7] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[8] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[9] A. Snavely and D. Tullsen. 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)*, pages 234–244, Cambridge, Massachusetts, Nov. 2000.

[10] M. Squillante and E. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, Feb. 1993.

[11] G. Suh, S. Devadas, and L. Rudloph. Analytical Cache Models with Applications to Cache Partitioning. In *Proc. of the 15th ACM International Conference on Supercomputing (ICS'01)*, pages 1–12, Sorrento, Italy, June 2001.

[12] G. Suh, L. Rudolph, and S. Devadas. Effects of Memory Performance on Parallel Job Scheduling. In *Proc. of the 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'02)*, pages 116–132, Edinburgh, Scotland, June 2002.

[13] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, Feb. 1995.

[14] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proc. of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 159–166, Litchfield Park, Arizona, Dec. 1989.

[15] R. Vaswani and J. Zahorjan. 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)*, pages 26–40, Pacific Grove, California, Oct. 1991.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

[17] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of the SuperComputing 1996 Conference (SC96)*, Pittsburgh, USA, November 1996.