1 Scheduling Algorithms with Bus Bandwidth Considerations for SMPs

CHRISTOS D. ANTONOPOULOS[1], DIMITRIOS S. NIKOLOPOULOS[2] and THEODORE S. PAPATHEODOROU[1]

[1]High Perf. Information Systems Lab, Computer Eng. & Informatics Dept., University of Patras, 26500 Patras, Greece

[2] Department of Computer Science, The College of William & Mary, 117 McGlothlin Street Hall, Williamsburg, VA 23187-8795, U.S.A.

1.1 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 most SMPs is the internal bus which is used to connect 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. The problem is more acute in low-cost, SMPs in which low-end, 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 taken only 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 are used to bias the scheduler, so that each thread keeps running on the same

processor. This helps threads build state in the caches without interference noise coming 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.

This chapter, presents a direct approach for coping 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, the slowdown due to bus saturation is even higher than the slowdown the programs would experience if they were simply time-shared on a subset of the system processors.

In this chapter we describe scheduling algorithms which address the problem directly. They select the applications to co-execute driven by the bandwidth requirements of their threads. Bus utilization information is collected from the performance monitoring counters which are provided by all modern processors. The algorithms measure the bandwidth consumption of each job at runtime. 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 processor. In other words, the scheduling 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%.

The rest of this chapter is organized as follows: Section 1.2 discusses related work. In section 1.3 we present an experimental evaluation of the impact of bus bandwidth saturation on system performance. In section 1.4 we describe the new, bus bandwidth-aware scheduling policies. Section 1.5 presents an experimental evaluation of the proposed algorithms in comparison with the standard Linux scheduler. Finally, section 1.6 concludes the chapter.

1.2 RELATED WORK

Processor scheduling policies for SMPs have been primarily driven by two factors: 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 that 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. If multiple jobs, including one or more parallel ones, run at the same time, space sharing schedulers prevent parallel jobs from using all processors of the system.

The effectiveness of cache affinity scheduling depends on a number of factors [16, 19, 21]. 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 if processors are time-shared among multiple threads. On the other hand, with short time quanta threads may not have enough time to build state on the caches.

Dynamic space sharing policies [11, 12, 20, 23] attempt to surpass the cache performance limitations by running parallel jobs on dedicated sets of processors, the size of which may vary at runtime. These policies tend to improve the cache performance of parallel jobs by achieving better locality, since jobs tend to execute on isolated sets of processors. The drawback of these policies is that they limit the degree of parallelism that the application can exploit. It has been shown that in most practical cases, the positive effect of improving locality outweighs the negative effect of losing processors. Thus, space-sharing policies tend to improve the performance of parallel jobs on multiprogrammed platforms.

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 [17, 18]. The common aspect of this work and the policies presented in this chapter 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 policies are using information collected from the program at runtime. 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 already proposed job scheduling algorithms for SMPs is driven by the effects of sharing system resources other than caches and processors. In particular, none of the policies is 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 considers 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, 15] has shown that the performance of the scheduler is improved when the scheduler takes into account the interference between applications on shared hardware resources. More specifically, it has been shown that it is possible to achieve better performance from multiprogrammed workloads, if the programs which are co-scheduled on multiple processors during a time quantum meet certain criteria that indicate good symbiosis between the programs on specific system resources. For example, the scheduler could select to co-schedule programs that achieve the least number of stall cycles on a shared functional unit of a multiple-issue processor, or achieve the highest utilization of instruction slots, or fit in RAM without incurring paging. These studies 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. Designed primarily for analyzing the performance of programs at the architectural level, 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 off-line performance analysis of applications either autonomously [7, 14, 24] or as the basis for building higher-level tools [3, 8]. N. Amato et al. define a performance prediction function which takes into account the memory hierarchy and contention effects [2]. The function is expressed in terms that can be attained using performance counters. The authors provide experimental evidence that it can be employed as a prediction tool by extrapolating performance counters measurements from small, pilot executions. However, information attained from performance monitoring counters has never been used before to either affect scheduling decisions at run-time on a real system, or drive run-time program optimizations.

1.3 THE IMPLICATIONS OF BUS BANDWIDTH ON APPLICATION PERFORMANCE

In this section we present experimental results which 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 [22]. The benchmarks have been compiled using the 7.1 version of Intel Fortran and C/C++ OpenMP compilers. We used codes which are 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, namely the medium which connects processors to memory, runs at 400 MHz. The operating system is Linux and the kernel version is 2.4.20. The values of hardware counters are monitored using the Mikael Pettersson's performance counter driver for Linux and the accompanying run-time library [13]. Unfortunately, the driver does not currently support concurrent execution of two threads on the same hyperthreaded 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 maximum sustained bandwidth measured by the STREAM benchmark [10] is 1797 MB/s when requests are issued from all processors. The highest bus transactions rate sustained by STREAM is 29.5 transactions/microsec, hence 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 a 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 as much as 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, then the third element and so on. The microbenchmark is programmed in C, so the array is stored in memory row-wise. As a consequence, 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 result, the microbenchmark has almost 0% cache hit rate for the elements of the array. 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/microsec.

The fourth experiment set is identical with 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 transactions rate is 0.0037 transactions/microsec.

Figure 1.1 depicts the bus bandwidth consumption of each application, measured as the number of bus transactions per microsecond. The reported bus transactions

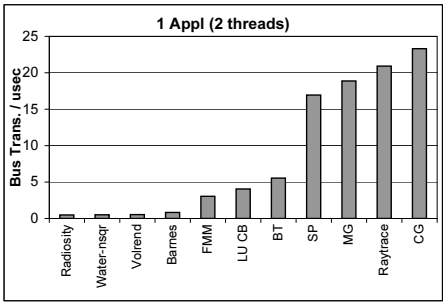

Fig. 1.1 Bus transactions rate for the applications studied, when the applications are executed alone, using two processors.

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 transactions 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.

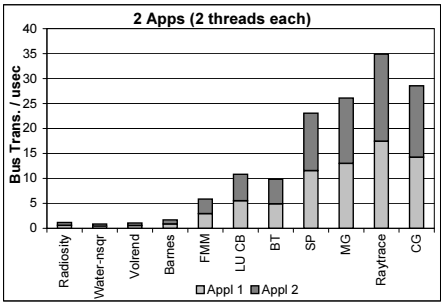

Fig. 1.2 Cumulative bus transactions rate when two instances of each application are executed simultaneously, using two processors each.

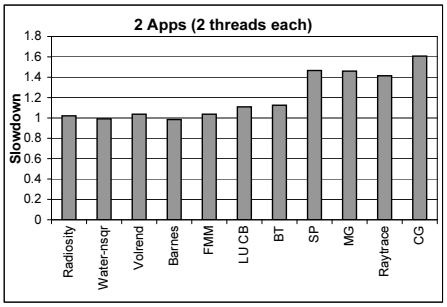

Fig. 1.3 Slowdown of the applications when two instances of each application are executed simultaneously, using two processors each. The slowdown in the diagram is the arithmetic mean of the slowdown of the two instances.

Figure 1.2 shows the accumulated number of transactions per microsecond, when two instances of each application run simultaneously using two processors each. The cummulative bus transactions rate is further analyzed in the diagram to depict the contribution of each application instance. Both instances contribute - as expected - almost equally, however the sustained bus transactions rate of each instance is generally lower than that of the standalone execution. 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.

It is worth noticing that four Raytrace threads yield a cumulative rate of 34.89 transactions/microsec, which is higher than the transactions rate achieved by four concurrently executing threads of STREAM (29.5 transactions/microsec). It has not been possible to reproduce this behavior with any other application or synthetic microbenchmark, even by varying the length of the data chunk transferred during each bus transaction.

Figure 1.3 shows the slowdown of the applications when two instances of each application are executed together using two processors each. Note that the two instances use different processors and there is no processor sharing. 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.

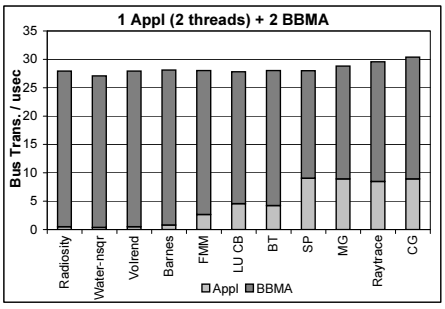

Fig. 1.4 Cumulative bus transactions rate when one instance of each application, using two processors, is executed together with two instances of the BBMA microbenchmark that issues back to back memory accesses without reusing data from the cache.

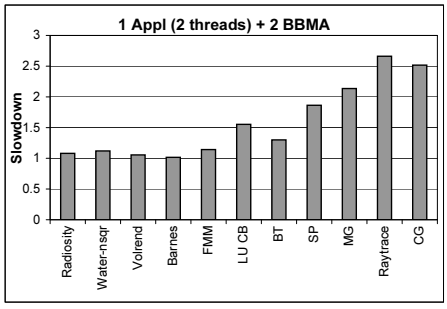

Fig. 1.5 Slowdown of the applications when one instance of each application, using two processors, is executed together with two instances of the BBMA microbenchmark that issues back to back memory accesses without reusing data from the cache.

Figures 1.4 and 1.5 illustrate the results from the experiments in which one parallel application competes with two copies of the BBMA microbenchmark that streams continuously data from memory without reusing them. These experiments isolate the impact of having applications run on an already saturated bus. Note that in figure 1.4 the bus bandwidth consumed from the workload is very close to the limit of saturation, averaging 28.34 transactions/microsec. Moreover, it is clear that the instances of the BBMA microbenchmark dominate the use of bus bandwidth. As a result, the available bandwidth for applications is often significantly lower, compared with their bandwidth requirements we measured during the standalone application execution. Memory-intensive applications suffer 2 to almost 3-fold slowdowns, despite the absence of any processor sharing. Even applications with moderate bus 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.

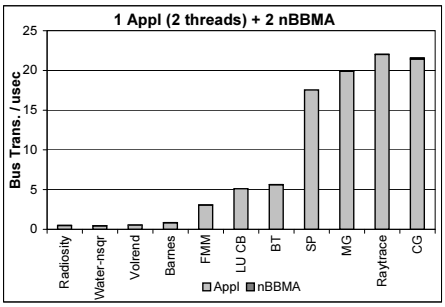

Fig. 1.6 Cumulative bus transactions rate when one instance of each application, using two processors, is executed together with two instances of the nBBMA microbenchmark that reuses data from the cache and does not consume any bus bandwidth.

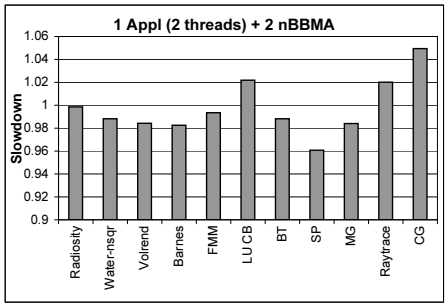

Fig. 1.7 Slowdown of the applications when one instance of each application, using two processors, is executed together with two instances of the nBBMA microbenchmark that reuses data from the cache and does not consume any bus bandwidth.

Figures 1.6 and 1.7 depict 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. In fact, the contribution of nBBMA instances to the totaly consumed bandwidth is not even visible in figure 1.6. This confirms that the slowdowns observed in the previously described experiments are not caused by lack of computational resources. Figures 1.6 and 1.7 also indicate that pairing high-bandwidth with low-bandwidth applications is a good way for the SMP scheduler to achieve higher throughput.

Figure 1.8 depicts the correlation between the reduction of the measured bus transactions rates of applications in all multiprogrammed executions and the corresponding slowndowns. All rate reductions and slowdowns have been calculated with respect to the standalone execution of the applications. The diagram also illustrates the regression line, fitted by applying the least squares algorithm on the (*bus trans. rate reduction, slowdown*) data points. Figure 1.8 indicates a close, almost linear relation between the limitation of the available bus bandwidth and the slowdown applications are expected to suffer.

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

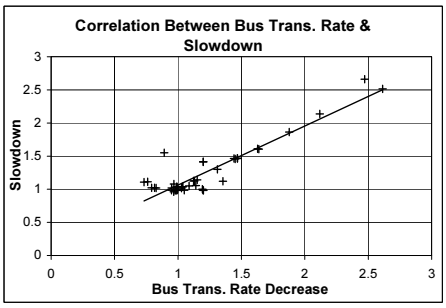

Fig. 1.8 Correlation between the reduction of the bus transactions rate and the slowdown of applications in the multiprogrammed executions.

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 these slowdowns it is reasonable to search for scheduling policies that improve application and system performance by carefully managing bus bandwidth.

1.4 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 are referred to as 'Latest Quantum Gang' (LQG) and 'Quanta Window Gang' (QWG). Both policies are gang scheduling-like, in the sense that, all threads of the same application are guaranteed to execute together. The scheduling quantum is fixed to a constant value. Applications coexisting in the system are conceptually organized as a list.

Table 1.1 outlines the pseudo-code of LQG. At the end of each scheduling quantum the scheduler updates the bus transactions rate statistics for all running jobs, using information provided by the applications. The bus transactions rate (BTR_{latest}) is calculated as the number of bus transactions caused by the application during the latest quantum, divided by the duration of the quantum. The previously running jobs are then transferred to the tail of the applications list.

Then, the policy initializes the Available Bus Transactions Rate ($ABTR$) for the next quantum to the System Bus Transactions Rate ($SBTR$). $ABTR$ quantifies the available bus bandwidth for allocation at any time during the scheduling. $SBTR$ is a constant, characteristic of the system bus capacity. Its value is equal to the maximum bus transactions rate that does not saturate the bus.

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

Foreach running application
$BTR_{latest} = \frac{Bus \quad Transactions}{Quantum \quad Length}$
Enqueue the application at the tail of available applications.
$ABTR = SBTR$
Unallocated Processors = System Processors
Allocate processors to application at the head of available applications and dequeue it.
$ABTR = ABTR - BTR_{latest}$
Unallocated Processors -= Application Threads
While *Unallocated Processors > 0*
$ABTR_{/proc} = \frac{ABTR}{Unallocated \quad Processors}$
Foreach available application
If *Application Threads* $\leq$ *Unallocated Processors*
$Fitness = \frac{1000}{1+|ABTR_{/proc} * Application \quad Threads - BTR_{latest}|}$
If no application with *Application Threads* < *Unallocated Processors* has been found
Scheduling has finished
Else
Allocate processors to the fittest application and dequeue it.
$ABTR = ABTR - BTR_{latest}$
Unallocated Processors -= Application Threads

Table 1.1 Pseudo-code of LQG algorithm. The QWG algorithm is similar.

Every time an application is selected for execution, its BTR_{latest} is subtracted from the available bus transactions rate ($ABTR$). Moreover, the scheduler calculates the available bus transaction rate per unallocated processor ($ABTR_{/proc}$) as

$$ABTR_{/proc} = \frac{ABTR}{Unallocated\, processors} \tag{1.1}$$

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

$$Fitness = \frac{1000}{1 + |ABTR_{/proc} * Application\, Threads - BTR_{latest}|} \tag{1.2}$$

Fitness is a metric of the proximity between the application's bus bandwidth requirements and the currently available bandwidth. The closer BTR_{latest} is to $ABTR_{/proc} * Application\, Threads$, 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, $ABTR_{/proc}$ turns negative and the application with the lowest BTR_{latest} becomes the fittest.

After each list traversal the fittest application is selected to execute during the next quantum. If there are still unallocated processors the $ABTR$ and $ABTR_{/proc}$ values are updated and a new list traversal is performed.

The QWG policy is quite similar to LQG. The sole difference is that instead of using the bus transactions rate of each application during the latest quantum, we calculate and use its bus transactions rate during a window of past quanta (BTR_{window}).

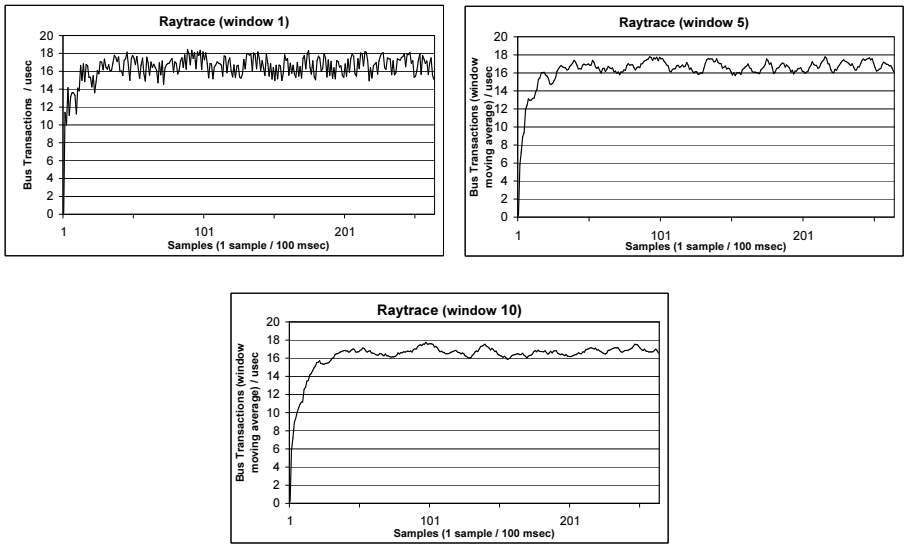

Fig. 1.9 Bus transactions rate of Raytrace when a window of length 1 (top, left), 5 (top, right), or 10 (bottom) is used. The reported rate is the average of the samples within the window. The hardware counters are sampled every 100 msec.

Using BTR_{window} instead of BTR_{latest} has an effect of smoothing sudden changes to the bus transactions caused by an application. This technique filters out sudden bursts with small duration, or bursts that can be attributed to random, external events such as cache state rebuild after a thread migration. However, at the same time it reduces the responsiveness of the scheduling policy to true changes in the bus bandwidth requirements of applications. The selection of the window length must take this tradeoff into account. Figure 1.9 illustrates the average bus transactions rate of Raytrace when windows of length 1 (no smoothing), 5 and 10 are used. When no filtering is applied, the bus transactions pattern is dominated by random variations of small duration. On the contrary, the use of a large window has the effect of distorting the observed pattern. The window used in QWG has been heuristically chosen to be 5 samples long. This window length limits 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.

The use of a user-level CPU manager facilitates the design, testing and comparison of scheduling policy without altering the OS kernel. We implemented a user-level CPU manager which executes as a server process on the target system. Its interface and functionality are similar to those of the NANOS CPU Manager [9].

Each application that wishes to use the new scheduling policies uses a standard UNIX-socket to send a 'connection' message to the CPU manager. The thread that contacted the CPU manager is the 'application leader'. The CPU manager responds to the 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 transactions rate information on the shared arena is expected to be updated. Moreover, the CPU manager adds the new application to a list of connected applications.

In order to ensure the timeliness of information provided from the applications, the bus transactions 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 applications are blocked / unblocked by the CPU manager according to the decisions of the effective scheduling policy. Blocking / unblocking of applications is achieved using standard unix signals. The CPU manager sends a signal to the 'application leader' which, in turn, is responsible to forward the signal to 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%.

1.5 EXPERIMENTAL EVALUATION

We have evaluated the effectiveness of our policies using three sets of heterogeneous workloads. Each experiment set is executed both on top of the standard Linux scheduler and with one of the new policies, using the CPU manager. All workloads have a multiprogramming degree equal to two. In other words, the concurrently active threads are 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 quantum of 100 msec, which resulted to an excessive

number of context switches. This is probably due to the lack of synchronization between the OS scheduler and the CPU manager, which in turn 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.

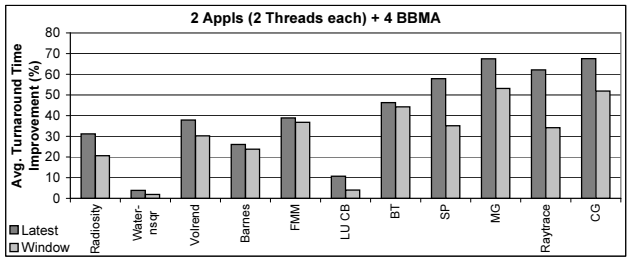

Fig. 1.10 Performance improvement (%) of the workloads when two instances of each application (using two processors each) are executed simultaneously with four instances of the BBMA microbenchmark. The reported values are the improvement in the arithmetic mean of the execution times of both application 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 1.10 illustrates the improvement each policy introduces on the average turnaround time of applications in comparison with 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 1.1). LQG achieves improvements ranging from 4% to 68% (41% in average). The improvements introduced by QWG 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 1.11 depicts the performance gains attained by the new scheduling policies.

LQG achieves up to 60% higher performance, however three applications slow down. The most severe case is that of Raytrace (19% slowdown). A detailed analysis

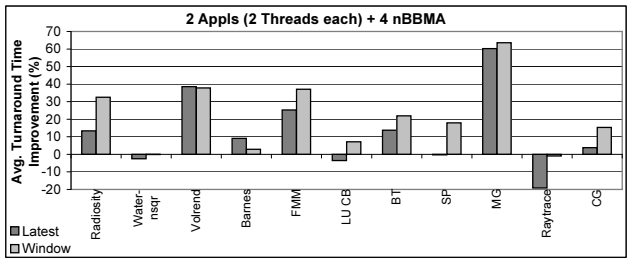

Fig. 1.11 Performance improvement (%) of the workloads when two instances of each application (using two processors each) are executed simultaneously with four instances of the nBBMA microbenchmark. The reported values are the improvement in the arithmetic mean of the execution times of both application instances.

of Raytrace revealed a highly irregular bus transactions pattern. The sensitivity of LQG to sudden changes of bandwidth consumption has probably led to this problematic behaviour. Moreover, from figure 1.1 one can deduce that running two threads of Raytrace together - which is the case due to the gang-like nature of our policies - may alone drive the bus close to saturation. LU CB and Water-nsqr also suffer minimal slowdowns due to their high sensitivity to thread migrations among processors. In average, LQG improved workload turnaround times by 13%. QWG turned out to be much more stable. It improved workload turnaround times by up to 64%. Raytrace slows down once again, however 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. Note that only one processor can transfer data over the shared bus at any given time snapshot.

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 performance improvements of the new scheduling policies over the standard Linux scheduler are depicted in figure 1.12.

LQG 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 QWG are 47% and 25% respectively. Two applications, namely Water-nsqr and LU suffer minimal slowdowns of 2% and 5%.

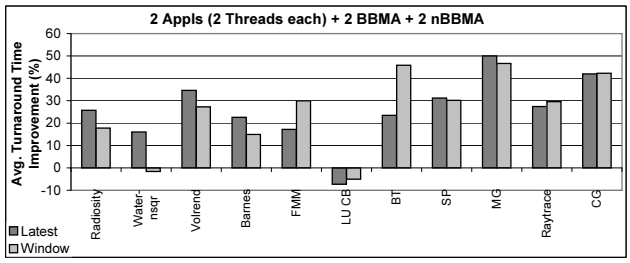

Fig. 1.12 Performance improvement (%) of the workloads when two instances of each application (using two processors each) are executed simultaneously with two instances of the BBMA and two instances of the nBBMA microbenchmark. The reported values are the improvement in the arithmetic mean of the execution times of both application instances.

In summary, for the purposes of this experimental evaluation we used applications with a variety of bus bandwidth demands. All three 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-bandwidth and low-bandwidth applications. As expected however, QWG proves to be much more stable than LQG. It performs well even in cases the latter exhibits problematic behaviour due to sudden, short-term changes in the bandwidth consumption of applications.

1.6 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 performance / price ratio in comparison with other architectures. 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 are optimized to minimize these effects.

In this chapter we 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 ranging 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 these counters have never before been used to improve application performance at run-time. LQG uses the bus transactions rate of applications during the latest quantum, whereas QWG uses

bus transactions rate calculated over a window of quanta. At any scheduling point both policies try to schedule the application with the bus transactions rate per thread that best matches the available bus transactions 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 non-demanding microbenchmarks. Both policies attained an average 26% performance improvement over the native Linux scheduler. Moreover, QWG has been much more stable than LQG. It maintained good performance even in corner-cases where LQG proved to be oversensitive to application peculiarities.

We plan to continue our study in the following directions. First, we plan to derive analytic or empirical models of the effect of sharing resources other than the CPU, including 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 also stress the bus bandwidth. This can be done in the context of scientific applications or other classes of applications, such as web servers and database servers. The policies can also be extended in the context of multithreading processors, where sharing occurs also at the level of internal processor resources, such as the functional units.

Acknowledgments

The first author is 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).

The second author is supported by NSF grants ITR/ACI-0312980 and CAREER/CCF-0346867.

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, April 1995.

2. Nancy M. Amato, Jack Perdue, Anrea Pietracaprina, Geppino Pucci, and Mark 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. Intel Corporation. Intel Vtune Performance Analyzer. http://developer.intel.com/software/products/vtune, 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. K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. *ACM SIGARCH Computer Architecture News, Proc. of the 25th Annual International Symposium on Computer Architecture (ISCA 98)*, 26(3), April 1998.

8. K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user Tools for Application Performance Analysis, Using Hardware Counters. In *Proceedings of the 15th International Conference on Parallel and Distributed Computing Systems (PDCS 2001)*, Dallas, USA, August 2001.

9. X. Martorell, J. Corbalan, D. S. Nikolopoulos, N. Navarro, E. D. Polychronopoulos, T. S. Papatheodorou and J. Labarta. A Tool to Schedule Parallel Applications on Multiprocessors. In *Proc. of the 6th IEEE Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'2000)*, volume 1911, pages 87–112, LNCS, May 2000.

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

11. 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.

12. T. Nguyen, R. Vaswani, and J. Zahorjan. Maximizing Speedup through Self-Tuning Processor Allocation. In *Proc. of the 10th IEEE International Parallel Processing Symposium (IPPS'96)*, pages 463–468, Honolulu, Hawaii, April 1996.

13. Mikael Pettersson. Perfctr performance counters driver for linux/x86 systems. http://www.csd.uu.se/∼mikpe/linux/perfctr.

14. A. Singhal and A. J. Goldberg. Architectural Support for Performance Tuning: A Case Study on the SPARCcenter 2000. *ACM SIGARCH Computer Architecture News, Proc. of the 21st Annual International Symposium on Computer Architecture (ISCA 94)*, 22(2), April 1994.

15. 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, November 2000.

16. 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, February 1993.

17. 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.

18. 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.

19. 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, February 1995.

20. 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, December 1989.

21. 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, October 1991.

22. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswider Pal Singh, and Anoop 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.

23. K. Yue and D. Lilja. An Effective Processor Allocation Strategy for Multiprogrammed Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1246–1258, December 1997.

24. Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of the SuperComputing 1996 Conference (SC96)*, Pittsburgh, USA, November 1996.