Scheduling Algorithms for Effective Thread Pairing on Hybrid Multiprocessors

Robert L. McGregor Christos D. Antonopoulos Dimitrios S. Nikolopoulos
Department of Computer Science
The College of William & Mary
Williamsburg, VA 23187-8795
rlmcgr,cda,dsn@cs.wm.edu

Abstract

With the latest high-end computing nodes combining shared-memory multiprocessing with hardware multithreading, new scheduling policies are necessary for workloads consisting of multithreaded applications. The use of hybrid multiprocessors presents schedulers with the problem of job pairing, i.e. deciding which specific jobs can share each processor with minimum performance penalty, by running on different execution contexts. Therefore, scheduling policies are expected to decide not only which job mix will execute simultaneously across the processors, but also which jobs can be combined within each processor.

This paper addresses the problem by introducing new scheduling policies that use run-time performance information to identify the best mix of threads to run across processors and within each processor. Scheduling of threads across processors is driven by the memory bandwidth utilization of the threads, whereas scheduling of threads within processors is driven by one of three metrics: bus transaction rate per thread, stall cycle rate per thread, or outermost level cache miss rate per thread. We have implemented and experimentally evaluated these policies on a real multiprocessor server with Intel Hyperthreaded processors.

The policy using bus transaction rate for thread pairing achieves an average 13.4% and a maximum 28.7% performance improvement over the Linux scheduler. The policy using stall cycle rate for thread pairing achieves an average 9.5% and a maximum 18.8% performance improvement. The average and maximum performance gains of the policy using cache miss rate for thread pairing are 7.2% and 23.6% respectively.

1. Introduction

Shared-memory multiprocessors are the most common building block for high-end clusters. The introduction of multithreading capabilities in mainstream microprocessors opens up new opportunities for boosting the performance of shared-memory multiprocessors by using hardware-supported multithreading for memory latency hiding and fine-grain parallel execution. These *hybrid* multiprocessors present new challenges to the system software, which needs to manage two levels of parallelism, intra- and inter-processor.

In most cases, spreading the threads of a job across many processors is a lot more efficient than clustering them in a few processors, due to contention for shared resources between threads on each processor. On the other hand, if the scheduler is flexible enough to pair[1] jobs in each multithreaded processor, it has a lot more opportunities for optimal utilization, but needs to consider the implications of pairing on the performance of jobs.

In this paper, we are concerned with the performance implications of sharing processor resources on multithreaded execution cores, and how these implications can be factored in a realistic job scheduler. Cache interference has so far been the only analogous criterion used in scheduling [18]. On multithreaded processors, looking only at the cache as a shared resource is insufficient. Multiple resources, including the cache, execution units, TLBs, branch predictors and others, are shared between threads and numerous complex interactions on these resources may affect performance.

We introduce and evaluate new scheduling policies for hybrid multiprocessors with simultaneous multithreaded processors [19]. The policies conceptually belong to a class of multiprocessor scheduling policies that analyze performance characteristics besides CPU utilization on the fly, to adjust and improve the schedule [2, 6]. One distinguishing aspect of the policies is that they use hardware event counters to collect performance metrics at run-time and optimize schedules driven by these metrics. In other words, hardware counters are used as an

1 The term *pairing* is used in a broad sense and implies running threads from multiple jobs in different hardware contexts and processors, rather than scheduling actual thread pairs.

online optimization tool, as opposed to offline optimization, which is their most common use. We also introduce a technique which allows to concurrently use the shared performance monitoring hardware by both execution contexts on Intel Hyperthreaded processors.

We use three run-time metrics that characterize thread interference within and across processors. One metric we consider is stall cycles, a single metric that characterizes the impact of sharing and interference on multiple shared resources of the processor. Cache misses at the outermost level of the cache hierarchy are an alternative metric we consider for tracking interference in processors, due to the sensitivity of many workloads to cache contention. Cache misses are also an important contributing factor to bus contention. We also consider bus transactions as an indication of contention between threads to access the bus interface on the processor. Contention for the bus interface increases memory latency. Driven from these metrics, we devise policies that attempt to optimize the utilization of memory bandwidth, and then balance a metric of choice (selected between stall cycle rate, cache miss rate or bus transaction rate) across processors, by strategically tuning the mixes of threads running on each processor. The policies are, to the best of our knowledge, the first policies aware of both multiprogramming and multiprocessing which have been tested on a real hybrid multiprocessor, outperforming the standard OS scheduler by up to 28.7%. Their complexity is linear with respect to the number of execution contexts and they can easily be embedded into general purpose priority schedulers.

The new policies have been incorporated in a user-level processor manager, implemented as a server for Linux systems. The latter is coupled with a run-time library that offers a communication interface between jobs and the processor manager.

The rest of the paper is organized as follows. Section 2 describes the processor manager. Section 3 summarizes the performance monitoring capabilities of Hyperthreaded Intel Pentium-4 processors and the technique we used to attain performance measurements in the presence of Hyperthreading. Section 4 describes the motivation and some quantitative intuition behind the policies we introduce in this paper. Section 5 outlines the design of the new scheduling policies and provides implementation details. Section 6 describes the experimental evaluation carried out to test the performance of the policies and discusses the results. Section 7 reviews related work. Finally, the paper is concluded in section 8.

2. Processor Manager

All policies presented in this paper were implemented in a low-overhead, user-level processor manager, which has its origins in earlier work of ours on the nanothreads programming model [11]. It has been heavily modified since then, to be used as a tool for experimenting with feedback-guided schedulers using hardware counters [2]. The processor manager allows for the design and testing of scheduling policies without having to alter the operating system.

The manager runs as a two-threaded server process on the target system. The first thread performs scheduling using regular time slicing, while the second thread undertakes the task of communicating with the applications and reacting to their requests. Each application that wishes to set itself under the control of the processor manager, sends a connection message for each one of its threads. The connection message and any other communication between applications and the processor manager is performed through a supporting run-time library. Besides notifying the processor manager of the creation of a thread, the connection call triggers initialization of whatever bookkeeping information the processor manager is using to schedule threads. For the policies presented in this paper for example, the processor manager initializes the hardware event counters needed to collect information for stall cycles, L3 cache misses and bus bandwidth consumption from each thread.

Applications may request a specific number of processors (or in the case of systems with multithreaded processors, execution contexts) from the processor manager. They may also communicate other information that the processor manager can use to improve its scheduling algorithms, such as a change in the scheduling status of threads during synchronization operations [1]. It is straightforward to incorporate these changes in the threading library used by the applications or, for more transparency, to enable them by intercepting library or system calls from an unmodified run-time library in the processor manager. In this work, we use the former option with a research OpenMP compiler [3]. Work on interfacing the processor manager with LAM/MPI and the binaries produced by Intel OpenMP Compilers is currently under way.

The main sources of overhead of the processor manager stem from messages for connecting, signals for blocking and unblocking threads, the manipulation of hardware performance counters, and naturally the execution of the scheduling policies. In any case the overall overhead of the processor manager is practically negligible. In order to quantify the overhead, we simulated the following worst-case scenario: The processor manager executes on the system, accepts thread connections and application requests, reads their performance counters and executes the scheduling policy. However, the policy decisions are not enforced. All threads are blocked at the end of each scheduling quantum and unblocked at the beginning of the next quantum. Moreover, threads are not bound to a specific execution context. As a result, the Linux scheduler is allowed to fully apply its own scheduling decisions. The execution time of the workloads in this setting is compared with their respective execution time when the processor manager is not activated. Under these circumstances, the overhead of the processor manager was measured to account for at most 1.8% of the execution time of applications in the workloads.

3. Performance Monitoring: The Case of Intel Hyperthreaded Processors

Most modern processors are equipped with extra hardware which allows monitoring of specific, performance related hardware events at run-time. These events characterize the interaction of applications with the hardware and the event counters provide a relatively cost-effective mechanism to attain such information. Typical events are related to the memory subsystem performance, the execution units utilization, branch predictions, resource stalls, etc. Traditionally, the hardware event counters have been used for off-line manual or automatic code restructuring and optimization. In the context of this paper we use them in a completely different way. We exploit them at run-time, in order to dynamically identify the characteristics of the execution environment and use this knowledge to reach educated scheduling decisions.

On Intel processors equipped with Hyperthreading capabilities, a single set of performance counters and the corresponding configuration registers are shared between two execution contexts. As a consequence, conflicts may arise if both threads on the same processor happen to request access to the same counter or configuration register. In order to avoid such conflicts, the operating system binds threads that have activated performance counter monitoring to the first execution context of each physical processor. In other words, at any time snapshot, only one execution context on each processor may accommodate threads that use performance counters.

In order to overcome this limitation, we have used two sets of counters for each event we measure. Both sets are activated by the processor manager on the thread which is specified to run on the first execution context of each physical processor during the next quantum. At the same time, we take advantage of a bitmask in the configuration registers which allows the programmer to distinguish between the events triggered by the thread running on each execution context. More specifically, in our case the first counter set is used to measure events caused by the thread running on the first execution context of the processor, whereas the second counter set measures events caused by the thread on the second execution context. At the end of time quanta, the processor manager reads the values from both counter sets. Given that the processor manager has accurate knowledge and complete control on the association between execution contexts and threads during any quantum, the values read from each counter set can be attributed to the corresponding thread. To the best of our knowledge, this is the first implementation which successfully deals with the problem of sharing performance monitoring hardware between threads on Intel Hyperthreaded processors. Furthermore, our solution does not require any modification to kernel code.

4. Motivation for Policies

This paper presents policies that belong to a class of multiprocessor schedulers which consider hardware resource interference as a primary scheduling criterion and track the run-time usage of hardware resources inside and outside processors to make informed scheduling decisions. The heuristics used in these policies may complement both conventional and customized schedulers, or can be used as user-level modules to enhance the job scheduling capabilities of front-end batch systems for supercomputers. The policies we introduce in this paper seek an optimal placement of simultaneously executing threads in the execution contexts of multithreaded processors with respect to certain interference-driven criteria. At the same time, they attempt to reduce the negative interference between groups of threads running on different processors due to cache-to-memory bandwidth saturation. We consider only CPU-intensive workloads running on hybrid multiprocessors. Ongoing work of ours is considering jobs with I/O and communication requirements.

Multithreaded processors make extensive use of resource sharing, as this yields more cost-effective designs. Sharing has both positive and negative sides. Threads that share data through the cache execute a lot faster than in conventional shared-memory multiprocessors, in which all data sharing happens through memory and the cache coherence protocol. On the other hand, if threads running on the same processor are not sharing any data, the effective cache space available to each thread is reduced because of interference from other threads. In the case of simultaneous multithreaded processors, co-scheduled threads must also share a single set of execution units and other resources, including TLBs, branch predictors, and instruction queues.

The strategy we investigate in this paper is to start with an educated placement of threads on processors obtained in a single time quantum. Following, we tune this placement on the fly via individual thread exchanges between multithreaded processors, or between mixes of threads co-scheduled during different time quanta on the same processor. Furthermore, we impose the following requirements: that performance is estimated using a few (more specifically two) run-time metrics obtained from hardware counters; that these metrics can be obtained simultaneously on each processor; and that no historical information other than some weighted averaging of old samples of the same metric is needed to make scheduling decisions. Another aspect of our strategy is that we seek information on interference inside and outside multithreaded processors, so that the scheduler can make a two-level decision. It first selects thread mixes that live well together on different processors and then splits these mixes into smaller groups, so that group members live well together on the same processor.

Tuning of pairings is triggered by imbalance in certain performance metrics across threads. The presence of imbalance in a given metric, corresponding to a shared resource, indicates that some threads exercise high pressure while others exercise low pressure on that resource. For resources with finite capacity that can meet the combined requirements of certain thread mixes, our strategy is to pair threads exercising high pressure with threads exercising low pressure. Intuitively, a low pressure thread leaves ample idle resources for high pressure threads and a high pressure thread does not leave enough resources for threads other than low-pressure threads.

In order to determine if imbalance in resource usage exists across applications and if this imbalance can be exploited by our scheduling policies, we gathered statistics for a sample workload consisting of four of the NAS parallel benchmarks written in OpenMP [7]: CG, FT, MG and SP. The data sets chosen are those of the class A problem sizes, and each instance of a benchmark uses two threads. Two instances of each benchmark were used in the workload. The workload runs in closed-system mode for seven minutes. This means that whenever a job finishes executing, a new instance of the same job is spawned and all execution contexts of all processors keep executing threads for the duration of the experiment. After seven minutes, the instances of the applications executing in the workload keep running until they finish. As applications finish, the degree of multiprogramming progressively decreases.

Our test system is a Dell PowerEdge 6650 Server with 4 Xeon processors using Intel’s Hyperthreading technology. Each processor runs at 2.0 GHz and has an 8 KB, four-way associative L1 data cache, a 12 KB eight-way associative L1 instruction trace cache, a 512 KB eight-way associative L2 cache, and an external 1 MB, eight-way associative L3 cache. The system has 2 GB of memory and runs the Linux 2.4.25 kernel.

Each Xeon MP processor with Hyperthreading technology has two execution contexts with private register files and instruction windows. The execution contexts share the caches, TLBs and execution units as they feed the processor’s superscalar pipeline. The experiment uses the Linux scheduler and no particular pairing strategy for the threads running on each processor. We present data on the number of bus transactions and stall cycles per microsecond.

Figures 1 and 2 depict those two metrics (bus transactions per microsecond per microsecond and stall cycles per microsecond respectively) and show the maximum, minimum and average value of the metrics across the eight execution contexts of the system. The values are sampled once per 100 ms. We calculate the moving average of the values over one second intervals (corresponding to windows of 10 samples) and plot this moving average over time. Figure 2 is plotted in logarithmic scale.

These figures pin-point imbalances within each sample. The imbalance ranges from a factor of five in bus trans-

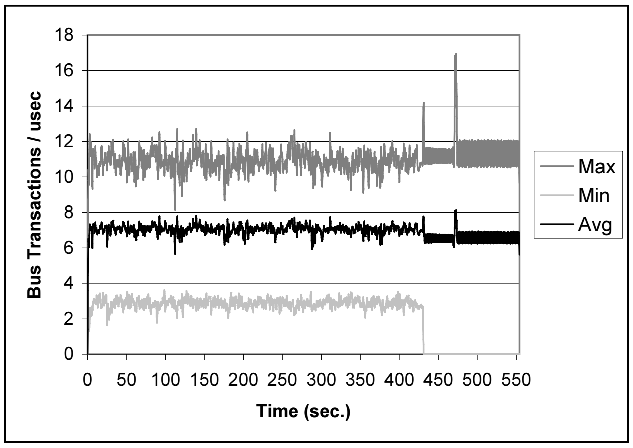

Figure 1. Maximum, minimum and average number of bus transactions per microsecond over time.

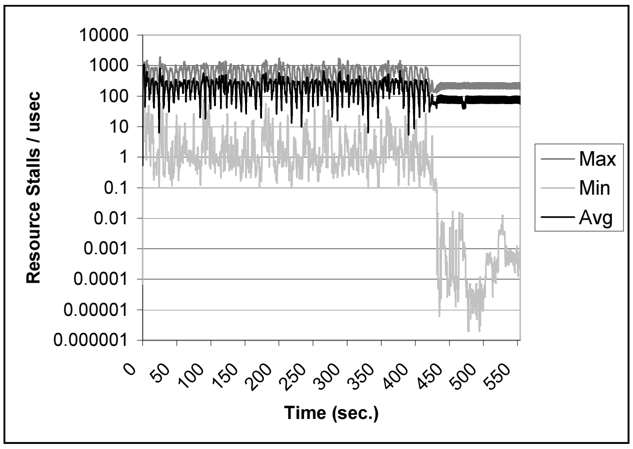

Figure 2. Maximum, minimum and average number of stall cycles per microsecond over time.

actions to 2-3 orders of magnitude in stall cycles. Similar trends were observed when we conducted the same experiment to measure imbalance in other metrics of the use of shared resources, such as L3 cache misses. In the presence of such imbalances, our policies try to reach an equilibrium by changing the placement of threads on execution contexts at run-time, so that threads which are sensitive to interference (e.g. threads with already high numbers of stall cycles or cache misses at a small temporal scale), are paired with threads that are causing insignificant interference (i.e. threads with small numbers of stall cycles or cache misses at the same scale). Regulating interference in this manner is likely to increase processor utilization and performance of individual jobs. Long idle periods of a thread due to excessive resource stalls may be overlapped with the activity of other threads that have lower requirements for the congested resources and are less dependent on them to make progress.

5. Policies

The scheduling policies we introduce use regular timeslicing with a base time quantum and co-schedule the threads of multithreaded jobs, in the spirit of gang scheduling. The goal of the policies is to select the best mix of threads to co-schedule during a quantum, by taking into consideration the interference between threads within and across processors. The policies focus on optimizing processor cycle usage for applications with high memory bandwidth requirements and sensitivity to contention for execution resources. Priority control is not enforced among the running applications[2], but it is easy to embed the policies into common priority schedulers. The policies do not take into account characteristics other than the interactions between threads sharing certain resources. They use detailed information from hardware counters as the driving factor in their scheduling decisions.

The processor manager uses a queue of waiting applications. Each application is served as a group of threads, so that all the threads of multithreaded applications are co-scheduled on different execution contexts, whenever they are allocated processor time. The first application to run during each time quantum is selected in a round-robin way, from the head of the applications queue. The round-robin allocation ensures that all applications will eventually have the chance to execute, independent of their performance-related characteristics. The free execution contexts, if any, are then filled with threads from other applications with the goal of optimizing bus bandwidth usage. The processor manager maintains historical information on the average bus bandwidth utilization requirements of the applications that executed during a window of previous quanta. This information is attained through performance monitoring counters. More specifically, the processor manager monitors the rate of bus transactions issued by each thread. While selecting the applications to execute during the next quantum, the scheduler attempts to identify and form a group of applications with bandwidth requirements close to that historic average. This way, the scheduler avoids to overcommit or waste bus bandwidth. At the same time, the scheduler is aparametric, in the sense that no *a-priori* knowledge is required concerning the maximum capacity of the system bus.

Note that, at this stage, the processor manager does not allocate specific execution contexts to each application. The new policies come into action during the second phase of scheduling, when the processor manager has already determined which applications to run during a time quantum. In that sense, the scheduling policy applied during the first phase for carefully managing bus bandwidth is orthogonal to the new polices: The former selects the threads to co-execute in the system during a quantum, while the latter focus on the optimal pairing of the selected threads on specific physical processors. The new policies are described in more detail in the following paragraphs.

The scheduling policies we introduce track imbalance in a metric (cache miss rates, stall cycle rates, or bus transaction rates) between threads or thread groups and use this imbalance as a criterion to pair threads with high rates of the specific metric with threads with low rates of the same metric. The policies use a single metric and event counter, which makes them appropriate for use in systems with limitations in simultaneous event counting.

The rest of the discussion assumes that the stall cycle rates is used as the primary metric. The scheduling policies that use cache miss rates and bus transaction rates are identical in all aspects except from the criterion used to tune thread pairings.

The base policy orders all threads that have been selected for execution by the number of stall cycles per time unit. This metric is calculated throughout a window of past quanta during which each thread has been executed. In our implementation and experimental evaluation the window was 5 quanta long. This window length has been chosen heuristically. We also experimented with using information from the latest quantum only. This strategy introduces noise in the performance data provided to the scheduling policies. Using the moving average in a window of previous quanta filters out this noise, while, at the same time, following accurately the bus transaction, stall cycle and cache miss rate patterns of threads.

Before deciding on thread placements, the scheduler must determine the number of processors on which threads

2 In common scheduling terminology, we assume that all jobs are CPU-intensive. The scope of this work is to improve service for such jobs in multithreaded/multiprocessor nodes used in high-end systems.

need to be paired. This will be equal to the number of physical processors any time the degree of multiprogramming is more than one. In other cases, we may be able to schedule some threads alone on certain processors. The number of pairs ($npairs$) is calculated using the following formula:

$$\begin{cases} 0 & \frac{nallocated}{nprocs} \leq 1 \\ nallocated \bmod nprocs & 1 < \frac{nallocated}{nprocs} < 2 \\ nprocs & \frac{nallocated}{nprocs} \geq 2 \end{cases}$$

where $nallocated$ equals the number of threads we have chosen to allocate during this time quantum and $nprocs$ is the total number of physical processors in the system.

Additionally, we define:

$$nsingles = nprocs - npairs$$

where $nsingles$ represents the number of threads that will be allocated alone on a physical processor. The policy allocates the $nsingles$ most demanding threads, i.e. the threads with the highest stall cycle rates, each to its own processor. We then place the remaining threads in pairs on the unallocated $npairs$ processors. At each iteration of our pairing algorithm, we place the unallocated thread with the lowest stall cycle rate and the unallocated thread with the highest stall cycle rate together, on the next available processor.

The policies try to pair threads from different applications on each processor. In general, if applications use more threads than the number of processors in the system, this constraint is not enforced. However, if applications require less processors the policies deliberately strive for pairing threads from different applications on the same processor. The reason is straightforward: the performance of a multi-threaded application when its threads run on different processors almost always exceeds the performance of the same application when the same number of threads runs on different execution contexts on the same processor. Contention for shared resources limits efficiency dramatically on current commercial multithreaded processors.

6. Experimental Results

We experimented with workloads composed of mixes of the NAS parallel benchmarks. We used the OpenMP implementations of the benchmarks [7]. The problem sizes of class A have been been chosen for testing. The problem sizes are substantial enough to yield realistic results, without overflowing the memory of our experimental platform while running multiprogram workloads with up to four benchmarks and 16 threads.

We used the processor manager with a base time quantum of 100 msec, equal to that of the standard Linux scheduler. Since our test system has eight execution contexts (two per physical processor), eight threads are selected to run during each quantum by the processor manager. All workloads run continuously for seven minutes in a closed system setting, as described in Section 4. We avoided using randomized workloads to ensure that our performance results were affected only by the processor scheduling policies used and the processor utilization they achieve.

Workload	Job mix
WL_1	CG(1,1), FT(1,1), MG(1,1), SP(1,2)
WL_2	BT(1,2), CG(1,1), EP(1,1), MG(1,1)
WL_3	CG(4,1), FT(4,1), MG(4,1), SP(4,1)
WL_4	BT(4,1), CG(4,1), EP(4,1), MG(4,1),
WL_5	CG(2,2) , FT(2,2), MG(2,2), SP(2,2)
WL_6	BT(2,2) , CG(2,2), EP(2,2), MG(2,2)

Table 1. Workloads used to test the scheduling policies. The notation (x, y) in parentheses is used to indicate that x copies of a benchmark, with y threads each are used in a workload.

Table 1 lists the workloads we used for our evaluation. The degree of multiprogramming is the total number of threads of the programs in the workload divided by the number of available execution contexts in the system. WL_1 and WL_2 have a multiprogramming degree less than one (0.625), and they are used to evaluate the thread pairing policies in isolation. In these workloads our schedulers use thread migration triggered by imbalances in L3 cache miss rates, stall cycle rates or bus transaction rates per thread per processor. There is no activation of the policy for regulating bus bandwidth consumption in these workloads. The workloads also isolate the effects of co-scheduling, since there is no time-sharing of execution contexts between threads. WL_3 and WL_4 use single-threaded benchmarks and their multiprogramming degree is two. Both regulation of bus bandwidth consumption and dynamic tuning of thread pairing are used in these workloads. WL_5 and WL_6 also yield a multiprogramming degree of two, however all benchmarks in these workloads use two threads. WL_5 and WL_6 are used to evaluate our policies when multiple parallel applications are co-scheduled on the same node.

The metric we are using to compare scheduling policies is the weighted mean execution time of jobs. Each job i is associated with a weight $w_i = \frac{n_i}{\sum_{i=1}^{N} n_i}$, where n_i is the number of instances of the specific job that finish execution in the seven-minute duration of the experiment and N is the total number of jobs. The weighted execution time is defined as $WT = \sum_{i=1}^{N} w_i t_i$, where t_i is the mean execution time of all instances of job i in the experiment. This metric factors in the widely varying length of different jobs, and accounts for penalizations or improvements that the schedul-

Workload	*Linux*	*Gang*	*Bus*	*Stall*	*Cache*
WL_1	11.91	11.07	11.39	10.98	11.56
WL_2	12.27	12.09	12.55	11.64	13.57
WL_3	34.99	36.63	30.00	34.04	30.72
WL_4	35.76	35.45	25.49	30.24	27.32
WL_5	19.18	18.74	16.40	17.79	18.70
WL_6	18.07	16.06	14.62	15.02	16.26

Table 2. Weighted mean execution times for the six workloads under different scheduling policies.

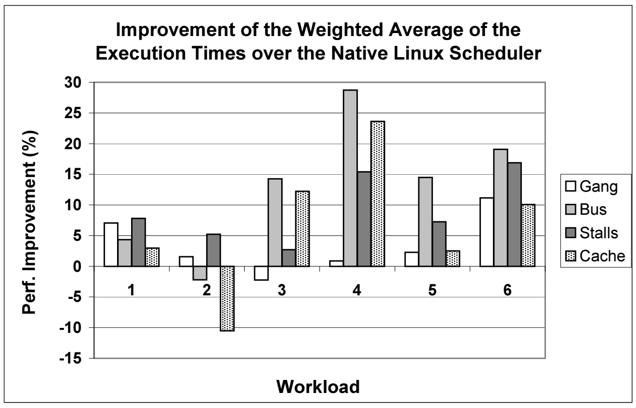

Figure 3. Improvements achieved by the new scheduling policies over the native Linux scheduler for each of the six workloads used in the experiments. A comparison with the simple gang scheduling is also included.

ing policies impose on different jobs, according to their requirements for shared resources.

Table 2 reports the weighted mean execution time for each of the six workloads, under five policies. In the table, *Linux* is used as an abbreviation of the native Linux scheduler; *Gang* represents a simple gang scheduling policy, without performance counters feedback or any specific thread pairing on processors; *Bus* represents the performance using the scheduling policy in which the bus bandwidth consumption from the threads scheduled in each time quantum is regulated and threads are paired on processors so that bus transaction rates are balanced between processors. In the same sense, *Stall* represents the thread pairing policy that balances stall cycle rates after regulating bus bandwidth consumption and *Cache* represents the thread pairing policy that balances L3 cache miss rates after regulating bus bandwidth consumption.

Figure 3 is a visual representation of the improvements achieved by our scheduling policies when compared with the native Linux scheduler. A performance comparison with the standard gang policy is also included. We observed solid performance improvements over the Linux scheduler in all cases, except WL_2, in which the policy that uses bus transaction rates is slightly inferior to Linux and the policy that uses L3 cache miss rates deteriorates performance by over 10%. The policy that pairs based on bus transaction rates is overall the most effective, yielding a 13.4% average improvement over Linux, with a maximum improvement of 28.7% for WL_4. This policy captures accurately the contention between threads for the bus interface, which subsumes contention because of read and write memory transactions, coherence, prefetching and I/O activity.

There is no consistent winner between the policy that pairs threads based on stall cycle rates and the policy that pairs threads based on L3 cache miss rates. The former achieves improvements ranging between 2.7% (WL_3) and 18.8% (WL_6) over the Linux scheduler. The latter achieves improvements ranging between 3.0% and 23.6% over the Linux scheduler, but incurs slowdown in one workload (WL_2).

L3 cache misses is a good indicator of conflicts because of sharing the cache between threads on the processor. However, this metric fails to capture contention accurately, because it does not reflect the impact of supporting multiple outstanding misses and overlapping their latency. A high L3 cache miss rate may be mitigated to some extent by this overlap. Furthermore, a raw number of cache misses does not provide a characterization of the misses.

Stall cycle rate appears to be a more effective metric, in the sense that it summarizes the impact of contention on all shared resources in the processor. This impact would be harder to assess efficiently if the scheduler needed to consider all the resources that can cause conflicts. The problem with this metric is that it subsumes delays because of contention and conflicts but it also subsumes delays because of inherent application behavior and does not differentiate between these two types of delays. Again, a characterization of stall cycles would assist the scheduler to make more educated decisions, but such a characterization is not possible without detailed hardware simulation.

Another trend observed in the charts is that the *Stall* policy tends to perform the best when the degree of multiprogramming is less than one. Workloads WL_1 and WL_2 do not impose time-sharing of execution contexts. The total number of threads running in the workloads is constantly less than the number of available execution contexts. This reduces both cache pressure and contention to access the bus interface on each processor. The former happens because fewer threads per processor cause less cache pollution and conflicts. The latter happens because with 8 execution contexts available for 6 threads, threads can be spread out between processors so that two processors run a single thread each, with no contention between their execution contexts. The properties of this workload make the policies using cache miss and bus transaction rates less effective.

The performance of the Linux scheduler can be explained as follows: Linux is oblivious to hardware metrics and schedules workloads based mainly on load balancing and dynamic priority criteria. More specifically, threads are organized in a global ready queue, shared among all execution contexts. Any execution context that finds itself idling tries to select the thread to execute next, by calculating the ”fitness” of all threads in the ready queue. Thread fitness depends highly on the dynamic priority of each thread. In an attempt to favor locality, the scheduler also gives a significant advantage to threads that executed in the past on the specific execution context and a smaller advantage to those belonging to the same virtual address space as the previously executing thread. That scheduling strategy is problematic in 2.4.x Linux kernels, because each execution context is considered as a single processor. Therefore, one idle execution context in a processor with a busy second execution context may be considered as ”unloaded” as an idle execution context on a processor with both execution contexts idle. This will lead the Linux scheduler to mistakenly schedule two threads on the same processor leaving another processor idle at times. This problem is solved in Linux 2.6 via the use of local queues per processor (i.e. per two execution contexts). In the case of our workloads, this problem shows up only in WL_1 and WL_2 when the multiprogramming degree is less than 1 and there are two idle execution contexts at any time in the system. The effect of this problem is not particularly harmful though. The Linux scheduler makes suboptimal scheduling decisions even when there are enough threads to keep all execution contexts busy and there is no need for load balancing.

It is more subtle to explain the relative behaviour of the *Stall* policy compared to the *Cache* policy. In the workloads with single-threaded applications (WL_3 and WL_4), the threads selected during each quantum exercise more pressure in the cache, than the threads selected during each quantum in the workloads with multithreaded applications (WL_5 and WL_6). In the latter case the threads have smaller memory footprints and less cache misses, since the applications are parallelized. As such, WL_3 and WL_4 are more sensitive to cache performance than WL_5 and WL_6. Therefore, they can be better optimized with a thread pairing criterion trying to alleviate cache pressure. On the other hand, stall cycle rates capture the impact of interference in resources other than the cache, which becomes relatively more prevalent in the workloads with multithreaded programs.

It is clear from figure 3 that even the simple gang scheduler outperforms the standard Linux scheduler in all cases with the exception of WL_3. Gang scheduling guarantees that all application threads will execute together, thus eliminating any potential delays during inter-thread synchronization due to inopportune scheduling decisions. All applications in WL_3 are single-threaded. As a result they can not benefit from gang scheduling. It is, however, worth noticing that the performance improvements attained by the three performance-driven policies are significantly higher than those attained by the simple gang policy. This comparison clearly isolates the benefits of the performance-driven, educated thread selection and pairing on the processors.

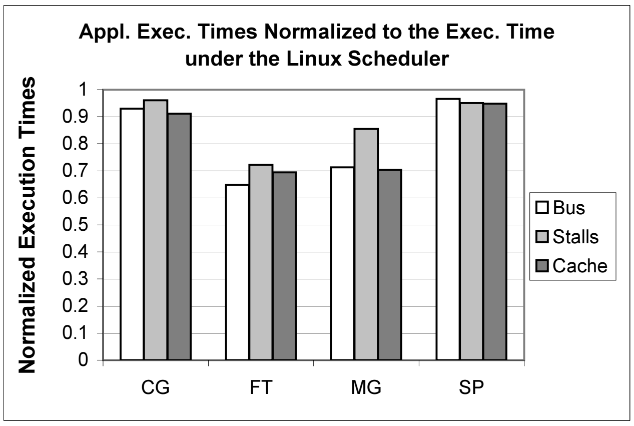

Figure 4. Average execution times of applications in the WL_3 workload under the new scheduling policies.

Figure 4 depicts the normalized, average execution times of each application participating in the WL_3 workload, when the workload is executed with the new scheduling policies. The average execution times have been normalized with respect to the corresponding execution time of the applications under the native Linux scheduler. The results suggest that by carefully selecting the co-scheduled threads and pairing them on physical processors, it is possible not only to reduce the average execution time of the workload but, at the same time, to improve the performance of each individual application.

7. Related Work

Prior research on scheduling algorithms that take into consideration the interaction between threads or processes in processors is closely related to the work presented in this paper. Several seminal works on cache affinity scheduling [18, 20, 15] used simulation, physical experimentation and analytical models to study the impact of time-sharing on cache performance. Heuristics for estimating cache affinity by measuring the time between consecutive executions of the same thread on the same processor are used in practically all shared-memory multiprocessor schedulers. Thread scheduling heuristics using accurate information on cache misses from the hardware have also been proposed for standalone multithreaded programs [10, 21].

The advent of simultaneous multithreading has stimulated research in all aspects of operating system support for processors using this technique [12]. Symbiotic job scheduling [13, 14], an idea proposed originally for scheduling on the Tera MTA architecture, has been proposed as a viable way for factoring the implications of sharing hardware resources between threads into the scheduling process. Symbiotic job scheduling relies on sampling the performance of different mixes of co-scheduled jobs, using different types of performance metrics, until a consensus on the best pairing is reached. This scheduling strategy may reach optimality for certain workloads but is very sensitive to their dynamic characteristics. The number of possible thread pairings grows quickly with the number of threads (or single-threaded jobs) in the workload and the number of execution contexts on the processor. The performance samples tend to get obsolete as jobs come and go, or when jobs move through different phases of their computation. The complexity grows even further if one needs to consider multiple hardware metrics to infer the interference in a given workload. Real processors can not support the simultaneous counting of many hardware events, therefore statistical sampling and averaging are necessary to gather all the information. This paper takes a different approach. Starting from a schedule selected to reduce interference between processors on the shared bus, we use hardware metrics to converge to a schedule that reduces thread interference within each processor.

Hardware mechanisms proposed to improve the quality of service to threads in terms of instruction throughput [5] relate closely to job scheduling issues and the policies we introduced in this paper, as they attempt to alleviate imbalances of service between threads by regulating the number of instruction issue slots used by each thread.

The interference between threads in the shared cache has also been a focal point of investigation for multithreaded processors and chip multiprocessors [9]. The problem presents several interesting challenges from the architectural perspective, but with respect to job scheduling, the most common technique used to alleviate thread interference is to partition the cache between processors or execution contexts [16, 17, 4, 8]. Most of these works propose static cache partitioning schemes, however dynamic schemes have also been proposed recently [8]. There is a consensus among these works that cache partitioning tends to be beneficial for job scheduling, however the problem of identifying the right static or dynamic partitions to use in the cache is difficult.

8. Conclusions

For hybrid multiprocessors built with multithreaded processors, it is necessary to consider both levels of parallelism, intraprocessor and interprocessor, while designing scheduling policies. This paper introduced new scheduling policies that collect at run-time information about hardware metrics related to the sharing of resources between threads. This information is used to tune the schedule by educated placement of threads on execution contexts. Tuning is motivated by the intuition that pairing jobs with high demands on shared resources with jobs with low demands on shared resources is likely to improve performance and resource utilization. Thread pairing decisions on each processor are made based on one of three types of information: bus transaction rate, cache miss rate, and stall cycle rate per thread. All these metrics provide insight into the performance and interaction of co-scheduled threads on a single processor. Our experiments indicate that bus transaction rates are more informative than stall cycle rates, which are in turn more informative than cache misses. The scheduling policies use no sampling or historical accumulation of the performance of specific pairings. They start with a random thread placement that tries to achieve fair sharing of processors and converge progressively to a better placement via educated decisions to migrate threads and alleviate contention.

We used workloads consisting of the NAS Parallel Benchmarks with two different degrees of multiprogramming and combinations of only sequential, only multithreaded, or both sequential and multithreaded jobs. Our evaluation focused on assessing the impact of the policies on job performance in the workloads. The policies we introduce achieved up to a 28.7% performance improvement over the native Linux SMP scheduler, and outperformed the Linux scheduler in 16 out of 18 cases and a performance oblivious gang scheduler in 13 out of 18 cases.

Ongoing work of ours addresses the problem of thread pairing for jobs that use both computation and communication. Early evaluation of thread pairing heuristics that take into consideration the role of each thread (communication or computation) revealed substantial (over 30%) performance improvements over the Linux scheduler. We are investigating ways to incorporate these heuristics in the processor manager and derive thread characterization automatically from hardware metrics. We also plan to investigate the impact of I/O and memory pressure in our future work. In parallel, we plan to adapt our processor manager to dynamically intercept library calls from third-party and OpenMP libraries, so that our scheduling policies become ubiquitous.

We also work on devising specifications of our policies for integration in mainstream priority schedulers.

Acknowledgments

This work is supported by an NSF ITR grant (ACI-0312980), an NSF CAREER award (CCF-0346867), an IST grant (2001-33071) and the College of William and Mary.

References

[1] C. Antonopoulos, D. Nikolopoulos, and T. Papatheodorou. Informing Algorithms for Efficient Scheduling of Synchronizing Threads on Multiprogrammed SMPs. In *Proc. of the 2001 International Conference on Parallel Processing (ICPP'01)*, pages 123–130, Valencia, Spain, Sept. 2001.

[2] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. In *Proc. of the 2003 International Conference on Parallel Processing*, pages 547–554, Kaohsiung, Taiwan, October 2003.

[3] E. Ayguadé, M. Gonzàlez, X. Martorell, J. Oliver, J. Labarta, and N. Navarro. NANOSCompiler: A Research Platform for OpenMP Extensions. In *Proc. of the First European Workshop on OpenMP*, pages 27–31, Lund, Sweden, Oct. 1999.

[4] G. Blelloch and P. Gibbons. Effectively Sharing a Cache Among Threads. In *Proc. of the 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'2004)*, Barcelona, Spain, June 2004.

[5] F. Cazorla, P. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable Performance in SMT Processors. In *Proceedings of the First ACM Conference on Computing Frontiers*, pages 433–443, Ischia, Italy, Apr. 2004.

[6] J. Corbalan, X. Martorell, and J. Labarta. Performance Driven Processor Allocation. In *Proc. of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI'2000)*, San Diego, California, Oct. 2000.

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

[8] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs. In *Proc. of the 10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 176–185, Madrid, Spain, Feb. 2004.

[9] B. Nayfeh and K. Olukotun. Exploring the Design Space for a Shared-Cache Multiprocessor. In *Proc. of the 21st International Conference on Computer Architecture*, pages 166–175, Chicago, Illinois, June 1994.

[10] J. Philbin, J. Edler, O. Anshus, C. Douglas, and K. Li. Thread Scheduling for Cache Locality. In *Proc. of the 7th International Conference on Architectural Suppport for Programming Langagues and Operating Systems (ASPLOS'VII)*, pages 60–71, Boston, MA, Oct. 1996.

[11] E. Polychronopoulos, X. Martorell, D. Nikolopoulos, J. Labarta, T. Papatheodorou, and N. Navarro. Kernel-Level Scheduling for the Nano-Threads Programming Model. In *Proc. of the 12th ACM International Conference on Supercomputing (ICS'98)*, pages 337–344, Melbourne, Australia, July 1998.

[12] J. Redstone, S. Eggers, and H. Levy. Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IX)*, pages 246–256, Cambridge, Massachusetts, Nov. 2000.

[13] A. Snavely and D. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Processor. In *Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, November 2000.

[14] A. Snavely, D. Tullsen, and G. Voelker. Sybmiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Proc. of the ACM 2002 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'2002)*, pages 66–76, Marina Del Rey, CA, June 2002.

[15] 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.

[16] G. Suh, S. Devadas, and L. Rudolph. 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.

[17] 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.

[18] 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.

[19] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *In Proceedings of the 22nd Intenational Symposium on Computer Architecture*, pages 392 – 403, June 1995.

[20] 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.

[21] B. Weissman. Performance Counters and State Sharing Annotations: A Unified Approach to Thread Locality. In *Proc. of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 127–138, San Jose, California, Oct. 1998.