Controlled Contention: Balancing Contention and Reservation in Multicore Application Scheduling

Jingjing Wang
Computer Science Department
Binghamton University
Email: jwang36@binghamton.edu

Nael Abu-Ghazaleh
CSE and ECE Depts
University of California, Riverside
Email: naelag@ucr.edu

Dmitry Ponomarev
Computer Science Department
Binghamton University
Email: dima@cs.binghamton.edu

***Abstract*—One of the benefits of multiprogramming in conventional systems is to allow effective use of resources. For example, when one application blocks for I/O, another can use the available CPU time, improving throughput and performance. In a multithreaded environment, contention for resources can lead to substantial interference between applications: an application with dependencies can suffer if a thread holding a critical dependency is not scheduled in time. In the HPC community, this problem is often addressed by reservation-based schedulers such as Gang scheduling. However, such schedulers cannot reap the benefits of resource multiplexing leading to underutilization of the resources and lower overall throughput of the system. In this paper, we explore the tradeoff between contention and reservation in multithreaded application scheduling on multicore systems. We show that neither approach is optimal under all conditions. We propose Controlled Contention (CC) — a scheduling algorithm that allows controlled contention for resources, allowing the benefits of contention while supporting limited reservation to reduce interference. CC provides around 25% improvement in relative speedup over the Completely Fair Scheduler (CFS). We also show that CC can significantly benefit from application-level interference management while providing fairness that is not possible to achieve with application-level adaptation alone. The combined approach (CC with application-level adaptation) provides an average of 21% improvement in throughput, 35% improvement in relative speedup and 36% reduction in energy for the application mixes we consider.**

***Keywords*-PDES, Multicore Application scheduling, Energy-efficient**

I. Introduction

With the end of Dennard's scaling, microprocessor manufacturers have resorted to multicore designs to effectively use the available transistor budgets. Today, commodity processing nodes at both the high-end (e.g., High Performance Computing and Data Center nodes) and the low-end (e.g., smart phones and tablets) use multicore CPUs. It is projected that the number of cores available on a chip will continue to increase, resulting in manycore architectures [2]. In fact, manycore chips are already being offered commercially for special purpose applications [29] and accelerators [24].

A. Reservation vs Contention Scheduling for Multicore Systems

As the number of cores continues to increase, multicore systems are becoming similar to full-fledged parallel computing machines with tight memory integration. At the same time, the Operating System (OS) on these machines is based on traditional Linux or Windows kernels. These OSes were not originally designed for such high performance environments. In particular, they use scheduling policies that allow unrestricted contention by applications for the available resources. This approach is inherited from single-threaded environments where it can significantly improve resource utilization. However, since multicores have an increasing proportion of multithreaded workloads, unconstrained contention can cause substantial interference when some application threads are not scheduled [20], [32], [26], [21]. The performance slowdowns resulting from interference vary depending on the characteristics of the application. However, they can far exceed the proportional slowdown typically seen with sequential applications. In particular, for tightly coupled applications with high dependencies, the slowdown due to interference can be dramatic [28].

The impact of interference on application performance is known in the parallel processing community. A typical solution to interference in HPC environments is to control application scheduling and resource allocation by coarse-grained resource reservation. In particular, one popular technique to control interference is gang-scheduling [11], which ensures that all the threads belonging to an application are co-scheduled together, without interference. Gang scheduling effectively provides time-multiplexing of the available resources among the competing applications, rather than among individual threads. Although interference is controlled, the coarse granularity of the resource allocation makes it impossible to dynamically multiplex the resources in situations where this multiplexing is beneficial. For example, an I/O-bound application can efficiently execute by running concurrently with a compute-bound application using the same resources. Conversely, providing the I/O bound application with an exclusive scheduling slice can lead to inefficiencies, as most of the cores would remain idle for the duration of the slice.

B. Proposed Approach: Controlled Contention

Effective scheduling for multicore platforms must balance two conflicting requirements: reservation to control interfer-

ence and contention to allow effective resource multiplexing. In this paper, we propose Controlled Contention (CC) — a scheduling algorithm that balances contention and reservation. Specifically, it allows multiple applications to be co-scheduled when they can tolerate interference. However, CC controls the degree of co-scheduling to limit interference and at the same time allow a healthy degree of competition for resources to improve utilization and application throughput.

A critical aspect of CC is the ability to learn the application behavior in order to determine which applications can be co-scheduled. Thus, a component of the framework monitors application behavior to estimate its resource demand profile. CC then takes the set of active applications and determines which ones can be co-scheduled together without causing destructive interference. Only subsets of applications that fit this profile are allowed to execute in each time slice. We describe the proposed approach in Section III.

C. Augmenting CC with Application-Level Adaptation

Recently, we proposed application-level adaptation for managing interference in environments with commodity OS schedulers that use contention. In particular, the application itself monitors interference, and if detected, remaps itself to use fewer threads while maintaining data locality [28] using a policy we call Locality-Aware Dynamic Mapping (LADM). For example, if interference is detected at two cores, the application deactivates two threads and remaps the work assigned to them to the other threads. The approach also detects the availability of extra resources and takes advantage of them when they become available. We demonstrated that LADM leads to effective management of interference when integrated with a Parallel Discrete Event Simulation (PDES) kernel [14]. While we used it for a specific application, the LADM pattern is general and can be applied to other applications, with different application-specific remapping policies.

LADM allows applications to adapt in the presence of uncontrolled interference. However, on its own, application-level scheduling can lead to significant unfairness, as adaptive applications relinquish the resources and non-adaptive ones continue to use them. In such situations, it is necessary to involve the OS to allow fair scheduling of the available resources. We also explore the combination of application-level scheduling with CC to further improve efficiency. The presence of some adaptive applications allows the scheduler to more aggressively schedule applications. In periods where interference arises, adaptive applications can locally reduce their resource demands and reach an effective schedule.

Conceptually, CC attempts to create separation in time such that at a given time, only a compatible set of applications are scheduled. In contrast, application-level adaptation creates separation in space (across resources) when unhealthy contention arises; adaptive applications reduce their resource footprint to avoid interference. It also allows CC to more aggressively increase contention relying on LADM to recover when the degree of contention proves unhealthy. We show that the presence of even a small number of adaptive applications can significantly improve performance of CC.

In summary, the contributions of this paper are:

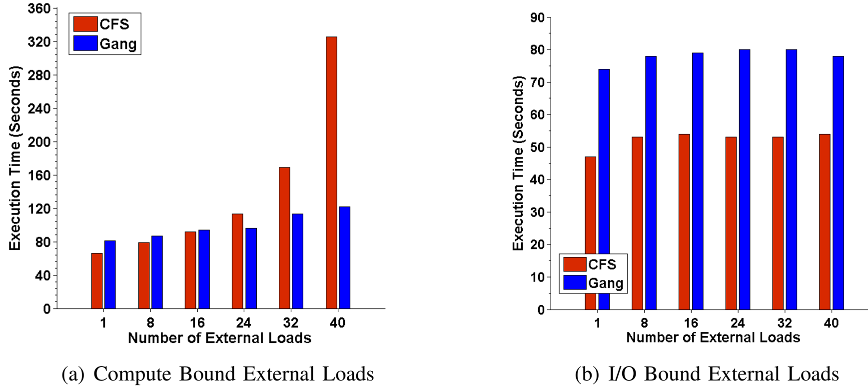
1) We propose CC — a scheduling algorithm that balances contention and reservation. CC improves resource utilization by allowing contention, but keeps this contention at a productive level by using reservation. CC identifies applications that can be co-scheduled and allows only such subgroups to be scheduled together in an OS time slice.
2) We compare the performance of CC to the Linux Completely Fair Scheduler (CFS), Gang Scheduling and CFS with application-level adaptation. Combining CC with LADM achieves an average of 35% improvement in relative speedup and 21% improvement in throughput compared to CFS, while improving fairness.
3) We evaluate the impact of CC and CC with LADM on the energy efficiency of the system and demonstrate an improvement of up to 36% for the application mixes we considered.

The rest of the paper is organized as follows. Section II discusses the limitations of CFS and Gang scheduling in terms of managing the interference. Section III presents the details of our approaches. In Section IV, we provide metrics to measure the performance of our scheduling schemes. Section V overviews the experimental setup, while Section VI presents experimental results. Finally, Section VII describes the related work, and Section VIII offers our concluding remarks.

II. Motivation

To illustrate the tradeoff between reservation and contention, Figure 1 shows the performance of the Parallel Discrete Event Simulation (PDES) application in the presence of interference from external loads. PDES is an important application with a fine-grained and dynamic dependency structure; as such it represents an example of application most sensitive to interference. In each experiment, we started a 48-way multithreaded simulation on an 48-core AMD Magny-Cours machine[1]. The simulation was executed concurrently with a different number of compute-bound external loads (Figure 1(a)), and I/O-bound external loads (Figure 1(b)) respectively. In particular, the compute-bound external load performs computation within a tight loop, while the I/O-bound external load repeatedly writes a string to a file. We experimented with both gang scheduling and CFS using SCHED_NORMAL scheduling policy. For gang scheduling,

[1]The AMD Magny-Cours consists of four AMD Opteron processors, with each having two six-core dies for a total of 48 cores. The processors are connected with custom AMD Hyper-Transport links. They experience NUMA delays both at the level of the caches (cores sharing the same die share a nearby tile of the cache) as well as the memory.



(a) Compute Bound External Loads

(b) I/O Bound External Loads

Figure 1. PDES Performance Under Interference

we used the Simple Linux Utility Resource Management (SLURM) [31] implementation.

As can be seen in Figure 1(a), when the interference caused by compute-bound external loads is limited (e.g. 1 external load), CFS achieves better performance than gang scheduling. This is because gang scheduling keeps some cores idle when the external loads are scheduled, leading to low CPU utilization. On the other hand, CFS attempts to maximize throughput by co-scheduling applications. When the contention on CPUs becomes high, however, it is more efficient to separate applications using gang scheduling. On the other hand, CFS can effectively co-schedule the PDES simulation with I/O-bound external loads, as shown in Figure 1(b).

III. Controlled Contention Scheduling

We have seen that reservation pessimistically separates applications even when they can effectively interleave the use of the resources. Under those circumstances, reservation slows down performance, reduces throughput and harms energy-efficiency. In contrast, contention can lead to significant interference, thus substantially harming performance. The goal of the proposed scheduler is to find the balance between contention and reservation in allocating resources in a multicore environment.

Starting from a contention-based scheduler such as the Linux CFS, CC scheduling introduces reservation in a way similar to that used by Gang Scheduling. Like Gang Scheduling, CC co-schedules all the threads of the application together. However, rather than scheduling one application in each slice, CC allows multiple applications to be co-scheduled when it determines that they can effectively co-exist. Thus, CC represents a hybrid approach, combining the strengths of reservation and contention.

Within this general philosophy, there are two primary components of CC: (1) Application characterization; (2) Scheduling decisions. We discuss these two components in the remainder of this section.

A. *Determining Application Resource Demands*

The degree of interference that is tolerated by an application depends on its nature. A highly-parallel application is generally resilient to interference, while an application with fine-grained dependencies can suffer from interference significantly. This is because critical dependencies could be held by threads that are context switched due to the lack of an available hardware execution context, preventing the whole application from making progress. The problem is exacerbated if applications use busy waiting synchronization primitives such as spin-locks or fuzzy barriers.

Rather than attempting to discern the internal behavior of an application to determine tolerance to interference, we focus on the application resource demand. Although resources include CPU time, memory, I/O bandwidth, and others, in this paper we focus on contention for processing time. The approach can generalize to other resources or combinations of resources. The scheduling decision can then examine the resource demands of the different applications to decide whether they can be co-scheduled in the same time slice.

We elected to monitor and characterize resource demands of applications during run-time, rather than through static analysis of off-line profiling. The resource usage of an application may change over time as it goes through different phases of its execution. Thus, it is important to continue monitoring to build an accurate estimate of resource usage. We use a background process for this purpose. The contention on CPUs can be reduced by separating applications with high CPU usages. To improve the system throughput, I/O-bound applications with low CPU usage can be co-scheduled with CPU-intensive applications.

To gauge the nature of an application from the standpoint of scheduling decisions, we define the weight of an application as the number of the application's threads (or processes) with high CPU demand. For example, if an application consists of 48 threads, but only 30 threads have high CPU usage, then the weight of this application is estimated as 30.

As a result, a CPU-intensive application is normally assigned a high weight, while another application with the same thread count but with an I/O bound behavior is assigned a low weight.

We use a simple policy to estimate the number of CPU-bound threads in an application. A thread is considered to have high CPU usage when its CPU usage exceeds a predefined threshold. We used the value of 20% based on our experimental observations. In other words, a thread that requires 20% or more of the CPU time is considered CPU-bound, whereas a thread with a CPU requirement of less than 20% is considered to be I/O-bound. To control the degree of contention, we allow the scheduler to decide the number of compute-bound threads it will tolerate using a threshold we call the Degree of Contention (or DC threshold), as explained below.

We note that this approach generalizes to other resources or combinations of resources. We also note that more accurate characterization of resource demands (e.g., using a more continuous classification of resource requirements) is possible and could lead to finer-grain control of the scheduling.

B. Making CC Scheduling Decisions

Given the estimates of the resource usage (CPU usage for our implementation), we now discuss how the scheduling decisions are made. The combined resource demand for a set of applications is obtained by summing the estimate of the demands of each individual application. Applications may be co-scheduled in the same time slice as long as their combined number of CPU-bound threads is below DC threshold which can be set to balance contention and reservation. If the DC value is low, contention is not accepted, and CC behaves similar to Gang scheduling. On the other hand, if DC is set very high, all applications can be co-scheduled and CC behaves similar to CFS. In the middle of these two extremes, a family of schedulers exist varying in the balance between contention and reservation.

For a given DC value, at each time slice, the scheduler selects a group of applications to run, where the sum of selected applications' weights is not greater than the value of DC. The selected applications are then assigned a time slice, and are co-scheduled. The behavior of the co-scheduled applications is monitored to update the estimate of their resource demand.

If some application completes before the assigned time slice expires, the scheduler looks for a candidate application from the list of suspended applications, with the purpose of maximizing the resource utilization. The sum of weights between the candidate application and all running applications is calculated. If the sum is below the value of DC, then the candidate application is scheduled for the rest of the time slice.

In addition, to prevent starvation, the scheduler maintains a counter for each application. The counter records the number of time slices that have been assigned to the corresponding application. When the scheduler searches for applications to schedule, the lookup starts from the application whose counter has the lowest value. The counter value is periodically aged to ensure that longer running applications are not unfairly treated when new applications arrive.

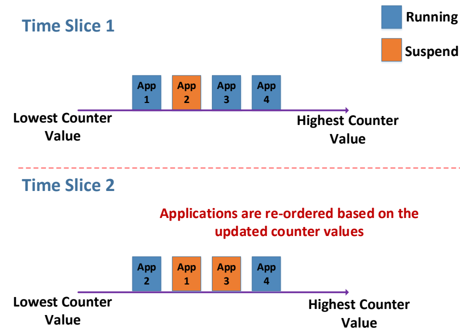


Figure 2. Example of Scheduling Four Applications

Figure 2 illustrates how CC schedules four applications. Recall that the background process periodically monitors the CPU usage of each application, and updates each application's weight. Suppose that the weights of the first, the second, the third and the fourth applications are 20, 43, 20, and 5 respectively. In addition, we assume that the value of DC is set as 48, and the counter of each application is set as 0 before the start of the first time slice. In this example, the first, the third and the fourth applications are co-executed during the first time slice. At the end of the first time slice, the counter of each of these three applications increases by 1. For the next slice, the second application has a higher priority to get scheduled and is scheduled with the fourth application in the next time-slice.

IV. Performance Metrics

In this section, we describe metrics that we use to evaluate the performance of the proposed schedulers. Specifically, we define three performance metrics: **Fairness**, **System Throughput**, and **Relative Speedup**.

A. Fairness

In OS schedulers, fairness in resource allocation among applications is a key design goal. Under fair operation and assuming flat priority, each application receives the amount of resources approximately proportional to its resource demands. As such, starvation can be prevented [18], ensuring that all co-located applications experience similar performance slowdown [10], [13]. On the other hand, low fairness leads to unfair resource allocation among applications. As a result, some applications experience worse performance slowdown than others.

We quantify the fairness by first calculating the relative slowdown of each application. An application j's relative slowdown S_j is obtained by dividing the execution time of

application j under scheduling by the execution time when it runs alone without interference. The fairness among n applications running concurrently is computed as the Jain's fairness index [15] (shown in Equation 1) applied to the relative slowdown. The value of the fairness ranges between 0 and 1, and higher value is better.

$$Fairness = \frac{(\sum_{j=1}^{n} S_j)^2}{n \times \sum_{j=1}^{n} S_j^2} \tag{1}$$

B. System Throughput

Fairness is not sufficient to measure the overall performance of co-located applications. As an example, gang scheduling can achieve high fairness among applications, but it results in low resource utilization [30] and higher slowdown. In this subsection, we define the second performance metric, called **System Throughput** (ST).

ST is defined as the number of completed applications per time unit. Thus, the higher values of ST mean that the applications are scheduled more efficiently. We derive ST from a definition proposed by Eyerman et al. [10]. Equation 2 shows ST for the situation with n applications running concurrently. Let $T_{inter,j}$ be the runtime of the application j under interference, and $T_{solo,j}$ be the runtime of the application executing solo j without interference. $\frac{T_{solo,j}}{T_{inter,j}}$ refers to the application j's fraction completed in solo runtime in the presence of interference. Effectively, this metric normalizes the progress of the application j under interference. ST is obtained by summing the normalized progress of each application.

$$ST = \sum_{j=1}^{n} \frac{T_{solo,j}}{T_{inter,j}} \tag{2}$$

C. Relative Speedup

Finally, to better compare the performance of different approaches, we propose another metric, called **Relative Speedup** (RS), to compare different scheduling approaches. RS is derived from the **Fair Speedup** metric proposed by Chang et al. [7]. As shown in Equation 3, the RS of one scheduling approach against another (the baseline scheme) is obtained by calculating the harmonic mean of each application's speedup (n applications in total). Higher RS values represent more efficient schedulers.

$$RS(approach) = \frac{n}{\sum_{j=1}^{n} \frac{T_{inter,j}(approach)}{T_{inter,j}(baseline)}} \tag{3}$$

V. Experimental Framework

Most experiments reported here were conducted on the 48-core AMD Magny-Cours machine described in the previous section. The remaining experiments were performed on a 16-core Intel SandyBridge machine to enable measurement of energy consumption. For the SandyBridge machine with Hyper-Threading enabled, each core can execute two simultaneous threads. In our experiments, we used ten applications with different characteristics. Eight of these benchmarks were chosen from PARSEC 3.0 benchmark suite [4]. PARSEC contains several multithreaded programs that have different computational profiles selected from various application domains. In addition, we use two flavors of PDES simulator: (1) FM: a baseline version which uses Fixed Mapping (FM) of the work to threads, and is highly susceptible to interference; and (2) LADM: a version that implements the Locality Aware Dynamic Mapping (LADM) policy that reduces resource demands when interference is detected.

For each application, we first identified its optimal thread configuration that achieved the best performance in the absence of interference on the Magny-Cours machine (as shown in Table I) and the Intel SandyBridge platform (as shown in Table II) respectively. We then used these configurations in our experiments.

VI. Performance Evaluation

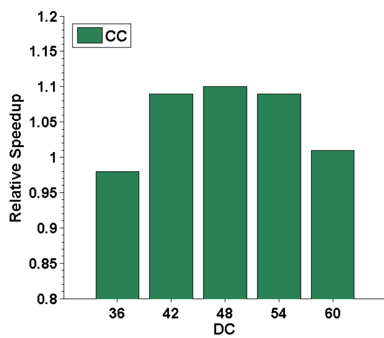
A. Evaluation of CC Scheduling

We first evaluate the performance of the CC scheduler on the Magny-Cours with different values of the DC threshold, which controls the degree of contention as described in Section III. In this first experiment, we ran 9 applications concurrently: FM and eight PARSEC benchmarks. We used these workloads to measure the relative speedup of CC against CFS. As shown in Figure 3, DC values of 48 and 29 achieve the best performance on the AMD Magny-Cours and the Intel SandyBridge platforms respectively. Thus, we used these respective thresholds in the remaining experiments.

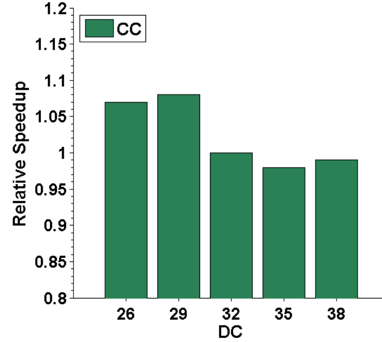
In the next experiment, we evaluate the performance of CC on the AMD Magny-Cours, as shown in Figure 4. In this experiment, we co-executed 2, 4, 6, 8, and 10 applications respectively. For the cases of 2, 4, 6, and 8 applications, each experiment consisted of 10 runs, with each having different group of applications. The applications in each run were randomly selected from 10 applications described in the previous section. Figure 4(a) shows the average system throughput for each approach, while Figure 4(b) shows the corresponding average fairness. Figure 4(c) shows the relative speedup of each approach against CFS. Gang scheduling exhibits the best fairness, but leads to poor system throughput. On the other hand, CC achieves 25% average improvement in relative speedup (and around 10% improvement in throughput) compared to CFS, with better fairness.

B. Effect of Application-Level Adaptation

Recall that application-level adaptation is an interference management approach which adjusts the number of active threads to the available cores and remaps the work accordingly [28]. LADM does not need scheduling support, but has two drawbacks: (1) applications that implement LADM suffer unfairness as they adapt their demands down,



(a) AMD Magny-Cours



(b) Intel SandyBridge Platform

Figure 3. Impact of DC

Table I
OPTIMAL THREAD COUNT AND CORRESPONDING RUNTIME WITHOUT INTERFERENCE ON THE MAGNY-COURS

Application	Optimal Thread Count	Runtime (seconds)
FM	48	33
LADM	48	33
Blackscholes	48	47
Bodytrack	48	35
Canneal	32	109
Dedup	8	8
Fluidanimate	32	58
Streamcluster	8	192
Swaptions	48	18
x264	32	13

Table II
OPTIMAL THREAD COUNT AND CORRESPONDING RUNTIME WITHOUT INTERFERENCE ON THE 16-CORE MACHINE

Application	Optimal Thread Count	Runtime (seconds)
FM	32	34
LADM	32	33
Blackscholes	24	74
Bodytrack	24	59
Canneal	24	159
Dedup	8	17
Fluidanimate	32	67
Streamcluster	24	90
Swaptions	32	38
x264	24	22

while non-adaptive applications do not; (2) applications need to be individually modified to use the LADM pattern. In contrast, an OS scheduling approach can accommodate all applications and does not rely on application cooperation. Thus, it is interesting to see if a scheduling approach can benefit from the support of application-level adaptation to provide better performance.

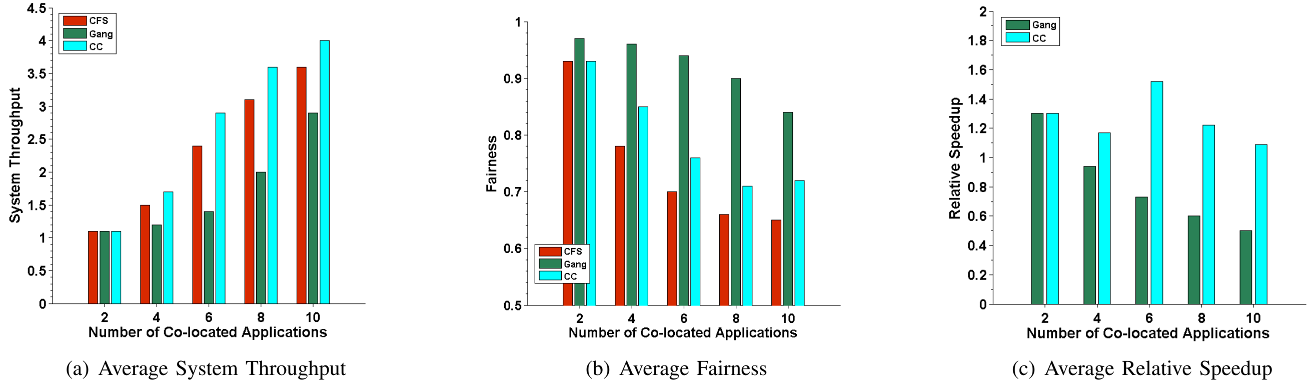
In the next experiment, we compare the performance of application-level adaptation (LADM) with that of CC. Note that in all the experiments that use LADM, only one application uses it (PDES); the other applications were not modified. The CC-only experiments use the FM PDES simulator without LADM, and kept other applications unchanged. Finally, we evaluate the performance of combining LADM with CC (we call this joint adaptation) to reap the benefits of both approaches.

As shown in Figure 5, LADM performs better than CC in the presence of low interference (e.g. two co-located applications), but becomes worse under high interference (e.g. eight co-located applications). In addition, the joint solution achieves the best performance for the case of 4, 6, 8, and 10 co-located applications. However, it performs slightly worse than LADM for the case of two co-located applications. The conclusion from these results is that under the limited interference it is sufficient to rely on application-level adaptation, while for higher degrees of interference the joint approach provides significant benefits. The joint approach provides an average of 35% improvement in relative speedup (across all workloads) and 21% improvement in throughput compared to CFS.

Figure 6(a), Figure 6(b), Figure 6(c) and Figure 6(d) show fairness and system throughput of each application group for the scenarios of 2, 4, 6, and 8 co-located applications respectively. In particular, we compare the performance of joint adaptation against using LADM alone and gang scheduling. The average fairness and system throughput are also plotted in each figure. Gang scheduling achieves the best fairness, but leads to poor system throughput. In addition, the performance gain of the joint approach increases with the degree of interference. For example, for the case of 8 co-located applications, the joint approach achieves 10% improvement in performance compared to LADM alone.

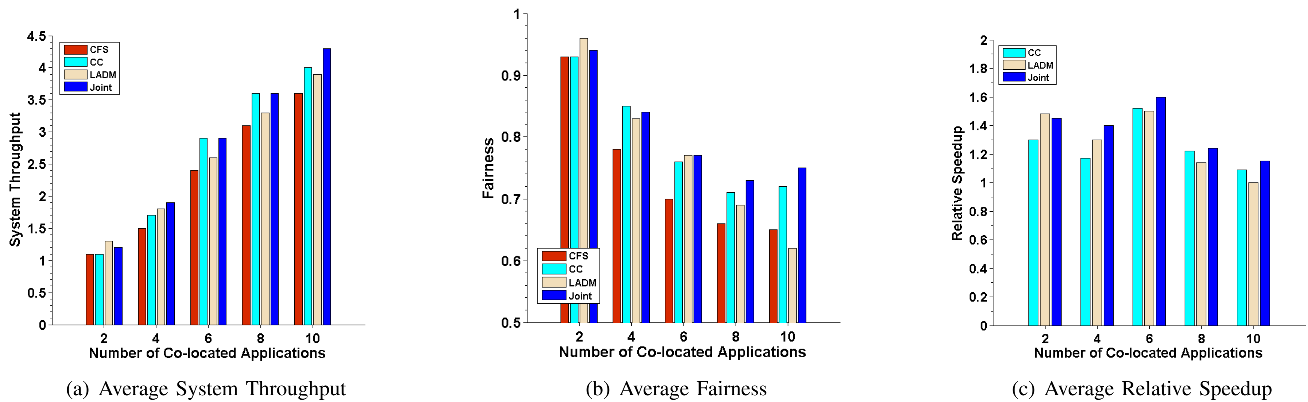
C. Impact on Energy Consumption

Finally, we evaluate the energy impact of each scheduling approach on the 16-core Intel SandyBridge platform, as shown in Figure 7. The platform provides an interface, called the Running Average Power Limit (*RAPL*), allowing the user to configure and read energy consumption of processors and



(a) Average System Throughput (b) Average Fairness (c) Average Relative Speedup

Figure 4. Performance of CC on the AMD Magny-Cours



(a) Average System Throughput (b) Average Fairness (c) Average Relative Speedup

Figure 5. Performance of Joint Adaptation on the AMD Magny-Cours

memory. In particular, we used a package called *likwid-powermeter* [25] to measure the energy consumption of the CPU cores and DRAMs during the execution of applications. In this experiment, we measured the energy consumption of each approach for the case of 2, 4, 6, 8, and 10 co-located applications respectively.

As shown in Figure 7, the joint approach provides up to 36% reduction in energy compared to CFS. CFS attempts to maximize the resource utilization without managing the interference. As a result, CFS causes longer execution time of applications, resulting in more energy consumption as well. Figure 7 also shows that gang scheduling consumes more energy than the joint approach. This is because gang scheduling leaves some cores idle while these idle cores also consume energy.

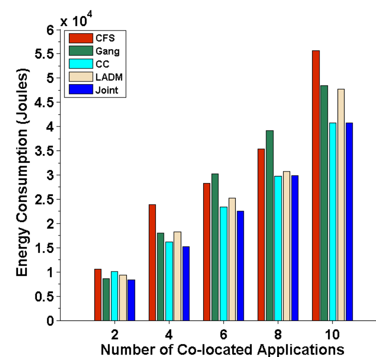
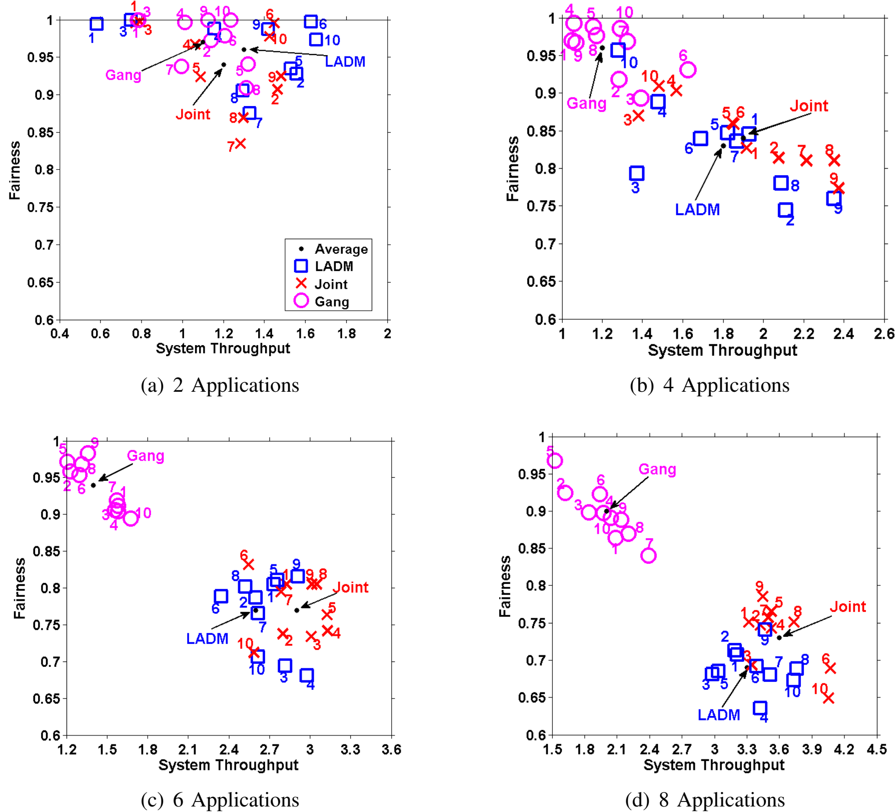


Figure 7. Energy Consumption on the Intel SandyBridge Platform

VII. Related Work

Shared memory parallel applications typically have dependencies between executing threads [28], [23]. Thus, when interference occurs, active threads have to wait for context switched ones before continuing to execute. The pace of execution can be substantially limited by critical path dependencies on suspended threads. As a result, the performance of these applications can be substantially harmed [20], [26], [21]. Although we focus on CPU interference, it is possible to consider interference on other resources such as I/O [9] or network bandwidth [6] simultaneously.

Two application-level approaches are widely used to balance workloads of threads at run-time: *work-sharing* and *work-stealing*. In work-sharing, when a thread completes its task, it grabs a new one from a central work pool shared



(a) 2 Applications

(b) 4 Applications

(c) 6 Applications

(d) 8 Applications

Figure 6. Performance of Each Application Group on the AMD Magny-Cours

across all threads [1]. In contrast, in work-stealing scheme, once a thread finishes its tasks, it steals other threads' tasks [12]. For example, Ribic et al. [22] proposed an energy-efficient work-stealing framework capable of adjusting the speed of each thread through Dynamic Voltage and Frequency Scaling (DVFS). As a result, a significant energy saving can be achieved with minimal performance loss. Though some applications benefit from work-stealing, it does not generalize to all applications; for example, for large PDES simulation, work stealing increases the critical path length of the simulation [27]. Work-sharing and work-stealing are load balancing approaches, thus neither approach can solve the interference problem unless a context switched thread does not hold any dependencies on partially executed tasks.

A. Scheduling Level Approaches

Some OS scheduling approaches have been proposed to manage interference. Gang scheduling [11] can mitigate the effect of interference by co-scheduling threads belonging to an application together. Conventional gang scheduling [11] separates applications in time to eliminate interference, however, this approach reduces system throughput and leads to poor performance of I/O-bound applications [17]. To increase system throughput, Wiseman et al. [30] proposed a paired gang scheduling approach where the threads of two jobs can be scheduled in the same gang as long as there are sufficient cores; Slurm implements this version of gang scheduling by allowing multiple applications as long as there are sufficient cores to accommodate all their threads without interference.

Zhuravlev et al. [32] proposed a scheduling algorithm to reduce the contention among applications on the shared last level of cache (LLC). The LLC miss rate of each application was monitored during its lifetime. The scheduler distributed applications across cores, ensuring that each shared cache has similar miss rate.

In the context of data centers, Delimitrou et al. [8] proposed an interference-aware scheduling algorithm for heterogeneous datacenters. After a short period of profiling, each incoming application was classified in terms of similarities with previously scheduled applications. A classification algorithm predicted the optimal server configuration capable of maximizing the performance of application. After classification, the applications were scheduled across servers, with the purpose of minimizing interference and increasing server utilization.

Kishore et al. [16] proposed a framework, called *ADAPT*, to co-schedule multi-threaded applications on multi-core platforms. The framework consisted of two allocators: core allocator and policy allocator. At run time, the resource

usage of each program was periodically monitored by a background process. The core allocator first predicts the performance loss of each program in the presence of interference under different core configurations with respect to memory access latencies, and then selected an appropriate thread-to-core assignment. After the core allocation, the policy allocator selected appropriate memory allocation and scheduling policies for each program. The framework is focused on memory locality rather than managing interference. We believe that this approach is complementary to CC and can be combined with it for additional performance benefits.

Bu et al. [5] developed a model to predict the performance of a MapReduce task under the contention of the CPU and I/O resources. In terms of this model, a MapReduce scheduling algorithm was developed to reduce the impact of interference among applications and to preserve task data locality. Bhadauria et al. [3] proposed a resource-aware co-scheduling algorithm that was capable of improving overall performance of applications and reducing power consumption. The key idea of this algorithm was to co-schedule applications having high resource consumption with ones having low resource consumption.

Mars et al. [19] proposed a co-scheduling algorithm on multi-core platforms. An application's sensitivity to interference was first characterized offline in terms of the difference between the application's IPC without interference and its IPC under interference. An application with high sensitivity was co-scheduled with the one having low sensitivity. In contrast, our approach is completely online. The work bears similarity in its attempt to allow compatible workloads to be co-scheduled. However, our approach in determining application compatibility and selecting co-scheduled applications is simultaneously more accurate and more flexible. Moreover, our work considers the possibility of combining application level adaptation with OS scheduling.

VIII. Concluding Remarks

In this paper, we examined the problem of scheduling for multi-core environments. Conventional schedulers such as the Linux CFS allow unlimited contention between active applications. In contrast, HPC schedulers use coarse-grained scheduling policies based on reservation to ensure that little contention for resources arises and no interference occurs. The basic tenet of our work is that controlled contention offers a more effective balance between improving resource utilization (through contention) and avoiding destructive interference (through reservation). We showed that this hybrid approach can outperform both pure contention and reservation, and can also provide higher energy-efficiency.

We also considered how CC interacts with a recently proposed application-level interference management approach (LADM) where applications reduce their resource demand when interference is detected. We discovered that even with one application out of ten supporting LADM, the combination of CC and LADM provides additional performance advantages while avoiding the weakness of using LADM alone.

Acknowledgements

This material is based on research partially sponsored by Air Force Research Laboratory under agreement number FA8750-11-2-0004 and by National Science Foundation grants CNS-0916323 and CNS-0958501. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies and endorsements, either expressed or implied, of Air Force Research Laboratory, National Science Foundation, or the U.S. Government.

References

[1] Greg R Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[3] Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for CMPS. In *International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA, 2010. ACM.

[4] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] Xiangping Bu, Jia Rao, and Cheng-zhong Xu. Interference and locality-aware task scheduling for MapReduce applications in virtual clusters. In *Intl. Symp. on High-performance Parallel and Distributed Computing*, HPDC '13, pages 227–238, New York, NY, USA, 2013. ACM.

[6] Marc Casas and Greg Bronevetsky. Active measurement of the impact of network switch utilization on application performance. In *Intl. Parallel and Distributed Processing Symposium*, pages 165–174, 2014.

[7] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 242–252, New York, NY, USA, 2007. ACM.

[8] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[9] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. Calciom: Mitigating I/O interference in HPC systems through cross-application coordination. In *Intl. Parallel and Distributed Processing Symp. (IPDPS)*, 2014.

[10] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, May 2008.

[11] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[12] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[13] Ron Gabor, Shlomo Weiss, and Avi Mendelson. Fairness enforcement in switch on event multithreading. *ACM Trans. Archit. Code Optim.*, 4(3), September 2007.

[14] Deepak Jagtap, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 520–531. IEEE, 2012.

[15] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared systems. *Technical Report, Digital Equipment Corporation, DEC-TR-301*, 1984.

[16] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Adapt: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, January 2013.

[17] Walter Lee, Matthew Frank, Victor Lee, Kenneth Mackenzie, and Larry Rudolph. Implications of I/O for gang scheduled workloads. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS '97, pages 215–237, London, UK, UK, 1997. Springer-Verlag.

[18] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. *SIGPLAN Not.*, 44(4):65–74, February 2009.

[19] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly characterizing cross core interference through contention synthesis. In *Intl. Conf. on High Perf. and Embedded Architectures and Compilers*, HiPEAC '11, pages 167–176, New York, NY, USA, 2011. ACM.

[20] Aroon Nataraj, Alan Morris, Allen D. Malony, Matthew Sottile, and Pete Beckman. The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, New York, NY, USA, 2007.

[21] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 55–, New York, NY, USA, 2003. ACM.

[22] Haris Ribic and David Yu. Energy-efficient work-stealing language runtimes. In *Proceedings of ASPLOS*, ASPLOS '14, pages 513–528, New York, NY, USA, 2014. ACM.

[23] Yu Su, Yi Wang, Gagan Agrawal, and Rajkumar Kettimuthu. SDQuery DSI: Integrating data management support with a wide area data transfer protocol. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 47:1–47:12, 2013.

[24] Tilera TILE64 processor, 2008. Documentation from Tilera Website http://www.tilera.com.

[25] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[26] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 303–312, New York, NY, USA, 2005. ACM.

[27] Stephen J. Turner. Models of computation for parallel discrete event simulation. *Journal of Systems Architecture*, pages 395–409, March 1998.

[28] Jingjing Wang, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Interference resilient PDES on multi-core systems: Towards proportional slowdown. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 115–126, New York, NY, USA, 2013. ACM.

[29] Yi Wang, Wei Jiang, and Gagan Agrawal. SciMATE: A novel mapreduce-like framework for multiple scientific data formats. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, pages 443–450, 2012.

[30] Yair Wiseman and Dror G. Feitelson. Paired gang scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 14(6):581–592, 2003.

[31] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple linux utility for resource management. In *Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[32] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM.