Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report Nr. 745

Systems Group, Department of Computer Science, ETH Zurich

Gang scheduling isn't worth it. . . yet

by

Simon Peter, Andrew Baumann, Zachary Anderson, Timothy Roscoe

November 25, 2011

Informatik
Computer Science

Gang scheduling isn't worth it. . . yet

Simon Peter[*], Andrew Baumann[†], Zachary Anderson[*], Timothy Roscoe[*]
[*] *Systems Group, Department of Computer Science, ETH Zurich*
[†] *Microsoft Research*

Abstract

The hardware trend toward higher core counts will likely result in a dynamic, bursty and interactive mix of parallel applications in personal and server computing. We investigate whether gang scheduling can provide performance benefits for applications in this scenario. We present a systematic study of the conditions under which gang scheduling might be better than classical general-purpose OS scheduling, and derive a set of necessary conditions on the workload.

We find that these conditions are rarely met today, except in a small subset of workloads, for which we give an example. However, we propose that this subset is potentially important in the future, if (for example) parallel algorithms become increasingly used for real-time computer-human interaction.

1 Introduction

In this paper, we examine the usefulness of gang scheduling for a general-purpose operating system on future multicore hardware. As parallel applications become commonplace in personal and server computing, general-purpose operating systems must handle a dynamic, bursty, and interactive mix of scalable parallel programs sharing a multicore machine [19, 23], for which current schedulers will likely be inadequate [10].

Many are looking to prior work in high-performance computing (HPC) to address the challenges of the multicore era. It is therefore important to critically examine to what extent such ideas can be transplanted to general-purpose computing. One example of an important idea from the HPC field is seen in *coscheduling* [22] and *gang scheduling* [9], which have long been proposed for scheduling synchronization-intensive parallel programs. These exploit lightweight *busy waiting* by ensuring that threads involved in a computation execute simultaneously on all cores. They are regarded as effective for programs employing fine-grain synchronization, but such programs typically have exclusive access to the machine (or partition thereof), and time slices are long (at least 100ms) relative to context-switch time.

In contrast, general-purpose OSes have not widely used gang schedulers, because parallel workloads that benefit from them have been rare, and techniques such as adaptive blocking synchronization are thought of as sufficient to achieve near-optimal performance [20].

In this paper we confirm that gang scheduling indeed has few benefits for current general-purpose workloads, except in a very limited set of cases. We characterize this space of workloads to show when gang scheduling can deliver significant benefits over classical OS schedulers.

One implication is that gang scheduling *might* be important in the future, if multiple parallel algorithms which synchronize at a fine grain are frequently invoked on small input datasets, for example for real-time computer-human interaction.

While our analysis applies to all workloads involving synchronization, we focus on parallel programs implemented using a single program, multiple data (SPMD) model, where data is partitioned among worker threads, each of which works on a partition in parallel. A control thread waits until all workers have finished processing, before the program can continue. The model is attractive because it allows the runtime to fix the number of parallel worker threads. Furthermore, it is easier to reason about than more dynamic models. Examples of the SPMD model are OpenMP [21], widely used in numerical libraries, and MapReduce [7].

We describe gang scheduling in Section 2, and in Section 3 present a systematic study of the SPMD workloads for which gang scheduling might be beneficial in a general-purpose OS. We conclude that it provides no benefits over current general-purpose schedulers, except when a set of narrow conditions are met, which we characterize. In Section 4 we give an example of such a scenario, based on existing algorithms retargeted at an interactive workload, and conclude with Section 5.

2 Background and related work

Originally introduced as *coscheduling* by Ousterhout [22], the class of techniques that we collectively refer to as *gang scheduling* includes several variants [1, 9, 11, 12]. All are based on the observation that a parallel job will achieve maximum overall progress if its component serial tasks execute simultaneously, and therefore aim to achieve this through explicit control of the global scheduling behavior. Compared to uncoordinated local scheduling on each processor, gang scheduling is particularly beneficial for programs that synchronize or communicate frequently, because it reduces the overhead of synchronization in two ways.

First, gang scheduling ensures that all tasks in a gang run at the same time and at the same pace. In parallel programs that use the SPMD model, work is typically partitioned equally among threads. This allows threads to reach synchronizations points at the same time.

Second, it is typically cheaper for a running task to resume from a period of *busy waiting*, in which it remains active without releasing the processor by *spinning*, than from *blocking* and voluntarily relinquishing control to the local scheduler, allowing another task to execute while it waits. However, if the task spins for a long time, it wastes processor cycles that could be used by other applications. Gang scheduling enables busy-waiting synchronization by reducing average wait times, and thus the time wasted in spinning.

The claimed benefits of gang scheduling include better control over resource allocation, and more predictable application performance by providing guaranteed scheduling of a job across all processor resources without interference from other applications. It achieves this without relying on other applications to altruistically block while waiting. Gang scheduling has also been proposed to enable new classes of applications using tightly-coupled fine-grained synchronization by eliminating blocking overheads [9].

Gang scheduling has not yet achieved uptake in general-purpose computing, where parallel synchronization-intensive applications are a niche, though notable exceptions include IRIX [4], which supported gang-scheduled process groups, and some user-level resource managers for cluster environments [16, 27]. Downsides of gang scheduling include underutilization of processors due to fragmentation when tasks in a gang block or the scheduler cannot fit gangs to all available cores in a time slice. The problem of *backfilling* these processors with other tasks is complex, much researched [6, 12, 22, 26], and beyond the scope of this paper.

The first thorough examination of the tradeoffs of gang scheduling performance benefits was conducted by Feitelson et al. [9] on a supercomputer with dedicated message passing hardware, and reports that gang scheduling improves application performance at a synchronization interval equivalent to a few hundred instructions on their system. We find that these tradeoffs still hold, albeit at a time scale equivalent to a hundred thousand cycles on our system, and thus the space has widened considerably. However, at the same time, time slices are shorter (15ms vs. 50ms on their system), which allows Linux to react more quickly to workload changes, and our results suggest that the space of workloads benefitting from gang scheduling is still small.

A more recent evaluation of gang scheduling appears as a case study [10], which uses a similar synthetic workload to ours, and compares to schedulers in several versions of Linux and Tru64 UNIX. It confirms that shorter time slices aid synchronization-oblivious schedulers, and shows that parallel applications tend to self-synchronize. We also confirm the results of this study.

Other recent work evaluates gang scheduling on clusters [8, 11] or supercomputers [13] and is less relevant to this study. Finally, some multiprocessor virtual machine monitors employ gang scheduling for performance and correctness reasons [14, 25], but VMs are a special case in that they use pure busy waiting – our results do not apply, since we assume waiting tasks eventually block.

3 Conditions for useful gang scheduling

Schedulers in general-purpose OSes like Linux or Windows typically offer best-effort scheduling with priorities. Feedback mechanisms prioritize IO-bound and interactive processes, and schedule the threads of a parallel process independently on each core. Several standard techniques can be applied with such schedulers to give good performance for parallel programs without the need for gang scheduling. We describe these techniques as they arise in our experiments, but note here that they are sufficiently commonplace and automatable to reflect the current state-of-the-art.

We have determined a set of necessary conditions for gang scheduling to enhance the performance of parallel applications over these techniques:

- fine-grained synchronization,
- good load balancing of work,
- latency-sensitive workloads,
- multiple important parallel workloads,
- a need to adapt under changing conditions.

In this section, we demonstrate the necessity of each condition using a microbenchmark that uses barrier synchronization among a pool of threads, and discuss the implications of these conditions.

```
void BARRIER(int interval, int variance)
{
  #pragma omp parallel
  for(;;) {
    int spincycles = interval + rand(variance);
    spin(spincycles);  // computation
    #pragma omp barrier
    iterations[omp_get_thread_num()]++;
  }
}
```

Figure 1: Pseudo-code of BARRIER program, with inputs for synchronization interval and maximum variance. `rand(a)` returns a normally-distributed random number in $[0, a]$. `spin(x)` exercises the CPU for x cycles.

In addition to these conditions, a few assumptions are needed for gang scheduling to make sense. First, the scheduler must be serving multiple competing tasks, otherwise the choice of scheduler has little impact on performance. Second, the application that might benefit from gang scheduling must be composed of multiple synchronizing/communicating execution contexts. If one thread never has to wait for another, it does not matter whether or not they are scheduled to run at the same time.

We also assume that blocking and unblocking a process incurs high costs for an individual application relative to spin-waiting, since much of the advantage realized by gang scheduling is achieved by avoiding the cost of blocking threads at synchronization calls. This assumption is currently realistic: on our Linux system, the cost of blocking (excluding the effects of TLB flushing and cache pollution) is approximately 7.4μs, whereas busy-waiting requires a tight loop of only a few instructions (40ns on average).

Finally, in the experiments we present, performance is highly sensitive to the implementation of the barrier synchronization primitive. A naive implementation hurts performance regardless of the scheduler, but gang scheduling is penalized more than others. Therefore, to give a "best-case" advantage to gang scheduling, we use our own implementation of a preemption-safe, fixed-time spin-block competitive barrier, described by Kontothanassis et al. [17], tuned to a spin time of 50μs. Using a balanced workload, this is enough to pass our barrier without blocking. Essentially, a good barrier implementation is also a necessary condition for gang scheduling to give a performance benefit.

3.1 Experimental environment

Our main workload for these experiments is BARRIER, a synthetic, synchronization-intensive parallel microbenchmark shown in Figure 1, run on a 16-core Supermicro H8QM3-2 4-socket AMD Opteron (Shanghai) server, clocked at 2.5GHz under Linux 2.6.32.

BARRIER counts iterations of a synchronized loop with configurable minimum interval (`interval`), and maximum variance (`variance`) of independent processing time in between synchronizations, to the arrival times of threads at synchronization points. When not explicitly stated, `interval` and `variance` are zero.

We use a simple user-space gang scheduler which forces Linux to schedule the desired threads by adjusting their priorities, similar to existing cluster gang scheduling solutions. Application performance under this scheduler is below the achievable performance of an in-kernel gang scheduler due to the overhead of running in user-space, using system calls to control program execution, but suffices for comparisons with other scheduling strategies. We have measured the overhead of our gang scheduler indirectly as less than 1%, by measuring iterations of a tight loop composed of two identical processes executing once under Linux and once under gang scheduling.

Our gang scheduler does not combat thread fragmentation due to an imperfect fit. However, in our experiments this property is not relevant to the results obtained: gangs always fit the available cores.

3.2 Fine-grained synchronization

We first show gang scheduling working when all conditions hold,[1] and that relaxing the requirement of fine-grained synchronization removes any benefits.

We run two BARRIER applications concurrently. In one, we vary the frequency of barrier invocation by spinning for a specific number of cycles on each thread before invocation using the `interval` parameter. We retain `interval` at zero for the other application, which acts as a competing background load contending for the processor cores.

Figure 2 shows the results: as synchronization granularity decreases, so do the benefits of gang scheduling since the overhead of blocking and waiting is amortized over fewer synchronization points. Despite this situation being highly favourable for gang scheduling, the performance benefit begins to diminish at a synchronization interval of 40μs, and is negligible when the synchronization interval is the average time-slice length of the Linux scheduler, in this case observed to be 15ms. This threshold can be pushed out slightly with a higher background workload, delivering a slight benefit with an interval up to 100ms on a heavily loaded system, but no further.

Few parallel applications meet this condition: since global synchronization does not scale, parallel algo-

[1] We assume that the computation in this benchmark has been generated in response to a changing workload.

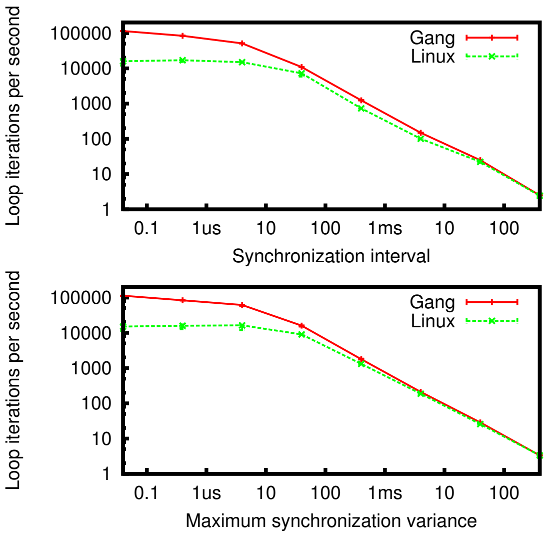

Figure 2: Gang scheduling vs. Linux under different synchronization granularities and barrier wait times. We show average/min/max over 10 runs.

rithms are designed to maximize the amount of unsynchronized parallel work, scaling it proportionally to the workload size. Thus, local computation dominates synchronization for most existing non-trivial workloads.

3.3 Good load-balancing of work

The average wait time for synchronization (the variance in synchronization times of different threads) must also be low for gang scheduling to offer performance benefits, otherwise it is better to block and allow other applications to run while waiting for synchronization. In the lower graph of Figure 2 we vary the `variance` parameter, so that each thread spins independently for a random number of cycles between 0 and `variance`.

As soon as the imbalance of work between threads reaches a few milliseconds of CPU time, gang scheduling loses effectiveness. Using different background loads or background blocking behavior does not lead to results significantly different from those in Figure 2.

3.4 Latency-sensitive workload

Gang scheduling time-slices parallel applications across multiple processors. As in classical time-sharing, this is only useful for applications with latency or responsiveness requirements. If the workload is entirely throughput-bound, and response time or completion time are not concerns, simple run-to-completion mechanisms (such as a first-come first-served batch scheduler) will suffice and offer lower overhead.

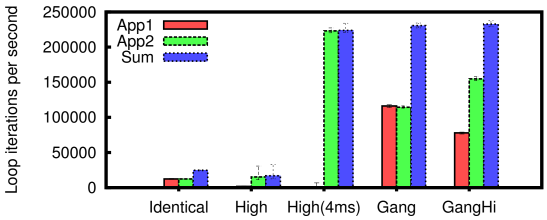

Figure 3: Gang scheduling vs. Linux under different scheduling priorities for App2.

We showed this experimentally by running two BARRIER programs, each with 16 threads, for 10 seconds in two configurations: as two 5-second, back-to-back batch jobs, and running concurrently with scheduling. We measured the achieved loop iterations per second. The batch configuration achieves higher throughput: 1.5% more iterations than with gang scheduling, due to reduced context switching and associated cache miss overheads (which are significant in our user-level gang scheduler). However, as a batch job the second BARRIER instance is only serviced 5 seconds after arrival.

This result is not surprising and, indeed, the overheads of time-slicing over batch scheduling have long been an accepted tradeoff in commodity computing, where interactivity and responsiveness are important metrics. However, it helps to re-emphasize that gang scheduling is only of benefit to workloads which are latency-sensitive, and adds the burden of a globally-coordinated schedule.

3.5 Multiple important parallel workloads

A necessary condition for gang scheduling to outperform priority-based scheduling is that the performance of *multiple* parallel applications is of interest. If not, simply prioritizing the one "important" application (e.g. the interactive program in the foreground) is sufficient to ensure that all its parts execute simultaneously.

We demonstrate this by running two BARRIER instances concurrently, each with 16 threads. Figure 3 shows the individual and aggregate performance of the two instances under various prioritization strategies. From left to right, we show the performance using the Linux scheduler with identical priorities (Identical), the Linux scheduler with App2 given a higher priority than App1 using the default spin time (High), and with a spin time of 4ms (High 4ms). These are followed by two versions of gang scheduling, one where the two applications are given equal length time slices (Gang), and one in which App2 is given a longer time-slice to simulate a higher priority (GangHi).

In the High experiment, App2 performs similarly to the Identical experiment, because spinning for only a short time causes threads to block frequently, rendering them unable to make use of their additional CPU allocation, while App1 is nearly starved due to frequent preemption by App2, resulting in worse overall CPU utilization. The performance of App2 is improved when spin time is increased, until it approaches that of gang scheduling, as shown in the High 4ms experiment.

We conclude that prioritizing a single application is a viable alternative to gang scheduling, with the caveat that, depending on the application's blocking behavior, its priority may have no effect on its performance.

3.6 Bursty, adaptive workloads

If demands are static over long periods, and fit within the resource constraints of the system, then a fixed number of hardware execution contexts (i.e. cores) can be allocated to an application. Space-partitioning schedulers have been proposed to achieve this [3, 19]. The existence of resource guarantees, and the knowledge that workloads will not exceed the allocation allows space partitioning to provide many of the same benefits of gang scheduling. Indeed, if the workload granularity matches the available cores in the system, space partitioning yields better performance than gang scheduling, as it does not impose any context switching overhead.

However, if workloads are bursty, and it is not possible to over-provision CPUs to applications, then space-partitioning is not possible without frequent repartitioning, which can be expensive. Further, it is difficult to detect, and then re-provision, resources that are underutilized. In such cases, time slicing is still desirable, and gang scheduling may be a reasonable choice, as it can yield higher total utilization of the system than sharing-agnostic schedulers [15]. We therefore include a bursty workload as a necessary condition to benefit from gang scheduling.

Space partitioning may become more feasible in future systems with high numbers of cores: unused CPU time could simply be ignored, and cores over-provisioned. However, it is likely that such systems will be performance-asymmetric [18]; in this case, a small number of powerful cores would be shared by applications, while a large number of "wimpy" cores may be space-partitioned. Gang scheduling would therefore remain suitable for the powerful cores.

4 Future application workloads

In the preceding section, we showed how gang scheduling offers a performance benefit only when a set of restrictive conditions is satisfied. Indeed, we were not able to find a standard benchmark suite showing compelling improvements under gang scheduling.

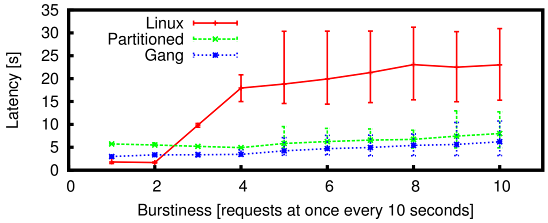

Figure 4: Parallel network monitoring under Linux vs. gang scheduling over a total run-time of 50s under different rates of burst.

In this section, we investigate the performance of gang scheduling on a hypothetical interactive workload that does satisfy the conditions enumerated in the previous section, and compare this performance with the default Linux scheduler. In particular, we consider an interactive network monitoring application that employs data stream clustering to analyze incoming TCP streams, looking for packets clustering around a specific attribute suggesting an attack. The stream clustering component of our application is drawn from the PARSEC benchmarking suite [5], which is designed to be representative of future applications for multicore computers, namely the widely anticipated, computationally-intensive and potentially interactive "recognition, mining, and synthesis" (RMS) workloads [2, 5, 24]. We argue that such workloads will become increasingly important and useful.

We must first verify that this workload does indeed satisfy the conditions we laid out. First, an interactive application will use a small workload size, which in the PARSEC stream cluster benchmark leads to more fine-grained synchronization that is still balanced. We confirmed this experimentally: When run for 3 seconds on a small workload (specifically, the "simlarge" dataset), the benchmark executes 16,200 barriers with an average processing time of 81μs, and average barrier wait time of 88μs, falling within the ranges measured in Section 3.2. The workload is latency-sensitive, since it is in the context of an interactive application. As multiple streams are being analyzed concurrently, there are multiple tasks of equal importance, and finally, since streams may arrive in bursts, the resource allocation must adapt to changing conditions. We used our tuned barrier implementation, rather than the naive default shipped with PARSEC.

The results of this application benchmark are presented in Figure 4. In this workload, for each of 4 TCP streams, cluster centers are updated every 16,000 packets, using 8 threads for each parallel computation. The graph shows the latency between the arrival of all 16,000 packets on a stream, and the update of the stream's clusters as the burstiness of requests increases. Gang scheduling allows the application to cope with increased burstiness, maintaining an average compute time of 5 seconds with a small variance. Performance under the Linux scheduler degrades as burstiness increases. Furthermore, variance in performance increases considerably. Gang scheduling also outperforms a static partitioning of the machine, because in order to handle the maximum burst rate, only 4 cores could be provisioned for each stream.

5 Conclusion

As we enter an era where massively-parallel computers are mainstream, rather than restricted to scientific computation, it is important to understand how to apply the insights of the HPC community appropriately to the somewhat different world of general-purpose computing.

While gang scheduling promises performance improvements for synchronizing parallel applications, we find that a long list of conditions must be met before it is useful in practice; for current general-purpose workloads, these preclude any benefit from the technique.

Nevertheless, the latency constraints and burstiness faced by parallel interactive applications force them to synchronize frequently as they await results from parallel computations to present to users. We hypothesize that future workloads are more likely to fall into this space.

References

[1] A. C. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, 2001.

[2] K. Asanović et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Dept., UC Berkeley, Dec. 2006.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. 3rd Symposium on Operating Systems Design and Implementation*, pages 45–58, 1999.

[4] J. M. Barton and N. Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In *Job Scheduling Strategies for Parallel Processing*, volume 949, pages 45–69. Springer-Verlag, 1995. Lecture Notes in Computer Science.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *17th international conference on parallel architectures and compilation techniques*, pages 72–81, 2008.

[6] J. Corbalan, X. Martorell, and J. Labarta. Improving gang scheduling through job performance analysis and malleability. In *Proc. 15th International Conference on Supercomputing*, pages 303–311, 2001.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. 6th Symposium on Operating Systems Design and Implementation*, pages 10–10, 2004.

[8] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proc. ACM SIGMETRICS*, pages 25–36, 1996.

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

[10] E. Frachtenberg and Y. Etsion. Hardware parallelism: Are operating systems ready? In *2nd Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 7–15, June 2006.

[11] E. Frachtenberg, D. G. Feitelson, J. Fernández, and F. Petrini. Parallel job scheduling under dynamic workloads. In *Proc. 9th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 208–227, 2003.

[12] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Transactions on Parallel Distributed Systems*, 16(11):1066–1077, 2005.

[13] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll. STORM: Lightning-fast resource management. In *Proc. 16th International Conference on Supercomputing*, 2002.

[14] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proc. 17th ACM Symposium on Operating System Principles*, pages 154–169, 1999.

[15] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. *SIGMETRICS Performance Evaluation Review*, 19(1):120–132, 1991.

[16] M. A. Jette. Expanding symmetric multiprocessor capability through gang scheduling. In *Proc. Workshop on Job Scheduling Strategies for Parallel Processing*, pages 199–216, 1998.

[17] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15:3–40, Feb. 1997.

[18] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. 36th International Symposium on Microarchitecture*, page 81, 2003.

[19] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proc. 1st USENIX Workshop on Hot Topics in Parallelism*, Mar. 2009.

[20] I. Molnar. Re: Gang scheduling in Linux. Linux-kernel mailing list, http://lkml.indiana.edu/hypermail/linux/kernel/0207.2/0407.html, July 2002.

[21] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 2008. Version 3.0.

[22] J. Ousterhout. Scheduling techniques for concurrent systems. In *IEEE Distributed Computer Systems*, 1982.

[23] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe. Design principles for end-to-end multicore schedulers. In *Proc. 2nd USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[24] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Peterson, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large scale CMP environment. In *Proc. EuroSys Conference*, Mar. 2007.

[25] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proc. 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 124–133, 2006.

[26] Y. Wiseman and D. Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):581–592, June 2003.

[27] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux utility for resource management. In *9th International Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60, June 2003.