Too Many Cooks in the Kitchen: Gang Scheduling for Predictable Performance

Matthew D. Fracis-Landau, Nathan T. Pemberton, and Sven Schwermer

{mfl, nathanp}@berkeley.edu, sven.schwermer@tu-harburg.de

December 16, 2014

Abstract

The high-performance computing (HPC) and cloud communities tend to take a very different approach to job scheduling. In the cloud, ease of use and interactivity are key. Typically, jobs are deployed on multi-core virtual machines. Each core of the VM is scheduled independently, leading to high variability in each thread's completion time. The HPC community requires predictable performance of every thread in order to minimize the time spent at barriers. To ensure consistency, HPC configurations typically use a batch manager which runs each job to completion before starting the next. This project seeks a middle ground, where programmers can rely on predictable performance of each thread (leading to efficient barriers) without having to wait for large jobs to finish before starting. We achieve this using gang scheduling, where all of a job's threads are started and ended together.

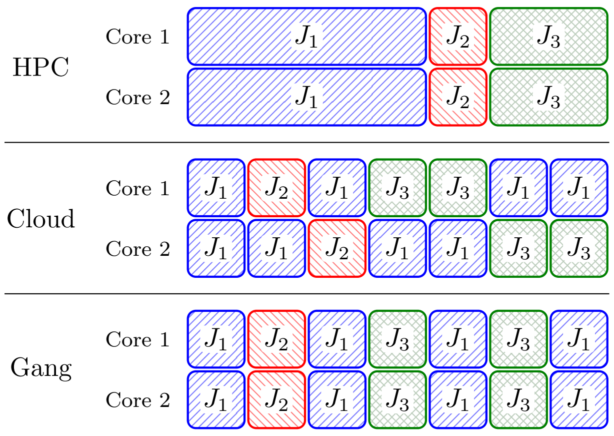
HPC
Core 1
Core 2
J_1 J_2 J_3
J_1 J_2 J_3
Cloud
Core 1
Core 2
J_1 J_2 J_1 J_3 J_3 J_1 J_1
J_1 J_1 J_2 J_1 J_1 J_3 J_3
Gang
Core 1
Core 2
J_1 J_2 J_1 J_3 J_1 J_3 J_1
J_1 J_2 J_1 J_3 J_1 J_3 J_1

Figure 1: Example schedules for different scheduling policies. HPC will run each job to completion before starting the next. Gang schedules all threads of a job simultaneously and cloud runs each thread independently without regard to job membership.

1 Introduction

Any sufficiently large system, for instance high-performance computing (HPC) or cloud, will involve some sharing of resources. However, the two communities tend to take a very different approach to sharing. In the cloud, ease of use and interactivity are key. Typically, jobs are deployed on multi-core virtual machines. This results in a number of mutually unaware schedulers running on top of each other. The virtual machine manager (VMM), guest OS, and runtime-library all make scheduling decisions independently. This introduces high variation in the run time of individual threads since they may be preempted independently of other threads. The HPC community, however, requires predictable performance of every thread in order to minimize the time spent in barriers. For example, if job 3 in Figure 1 had a barrier early on, the example cloud schedule would have caused core 0 to wait for core 1 to catch up before proceeding, causing significant wasted time. To avoid situations like this, supercomputers typically use batch-managers such as SLURM [16] or Torque [1] which run each job to completion (the HPC schedule in Figure 1). This maximizes the performance of each job, but may cause small jobs to wait behind larger jobs (job 2 in Figure 1).

This project seeks a middle ground, where programmers can rely on predictable performance of each thread (leading to efficient barriers) without having to wait for large jobs to finish before starting. We achieve this using gang scheduling, where all of a job's threads are started and ended together. In this case, both threads of job 3 will hit the barrier at the same time, but job 1 is able to finish in a timely manner.

Supercomputers have historically used gang-scheduling for this purpose with some success. For instance, Lawrence Livermore National Laboratory implemented gang scheduling on top of several different supercomputer environments and reported significant advantages [5]. However, these systems had tightly integrated operating environments and required significant re-engineering for each system. These systems were purely focused on scientific computing and would not be amenable to the modern cloud environment. Likewise, VMware has supported some form of co-scheduling in their vSphere product for most of it's lifetime [15]. This feature was mostly focused on preserving timing assumptions made by guest operating systems and to our knowledge has not been evaluated in the context of HPC workloads. We unify these environments by running HPC workloads on top of virtual machines using the Xen hypervisor [7]. In this environment, virtual machines (called domains in Xen terminology) run a single application, and users may submit new applications by creating new domains to run them. Section 3 has more details on our operating environment.

In section 2 we provide background on the issues of synchronization in large systems. Section 3 describes our implementation of gang scheduling on top of the Xen hypervisor and our experimental system. We then describe the benchmarks we chose in section 4 and then present our results in section 5. In section 6 we present some background and our experiences on sources of system noise and it's effects on the performance of parallel workloads. We then summarize related work in section 7 and future work in section 8. Finally we conclude in section 9.

2 Motivation and Background

A lack of synchronization between thread scheduling has been shown to cause a number of issues. Two prominent issues are that of lock-holder preemption (where guest OS threads may hold spinlocks for much longer than intended) and barrier latency.

2.1 Lock-Holder Preemption

Lock-holder preemption refers to the situation where a thread that holds a lock is preempted. Any other threads that are waiting on that lock may still be scheduled, wasting significant time. Typically, spinlocks are held by OS threads which are never preempted, but in a virtualized environment, the hypervisor may swap out virtual CPUs at any time. This can cause performance issues and may even cause the guest OS to perform incorrectly (if timeouts expire). There have been a number of attempts to deal with this issue (see section 7), but these approaches all assume that the only applications requiring fine-grained synchronization are operating systems, and therefore lock-holder preemption only occurs within operating system code. In fact, HPC applications frequently require very fine synchronization (on the order of 1–5ms) [11]. By ensuring that all threads are scheduled simultaneously, gang-scheduling naturally avoids the issues of lock-holder preemption without making any assumptions about guest behavior.

2.2 Barrier Latency

While debugging the performance of the ASCI Q supercomputer at Los Alamos National Lab (LANL), Petrini et al. found that barrier performance was critical to overall system performance [11]. They found that even small interruptions could cause significant problems in highly parallel machines such as the ASCI Q supercomputer with thousands of cores. In their analysis, they found that periodic tasks would independently interrupt processors. Even though the total amount of interruption was small (roughly 2.5% of the time was spent in "noise"), every processor had to wait for every instance of noise due to frequent, fine-grained barriers. One key result of this is that frequent small interruptions are worse than less-frequent large interruptions. By eliminating most noise, and consolidating the rest into fewer nodes, the authors were able to improve performance significantly.

Gang scheduling naturally addresses this issue by ensuring that all the threads that synchronize on the same barrier make similar progress. Smaller threads that cause noise are forced to run in the gaps between the gang-scheduled threads, thus consolidating noise into a small number of interruptions.

3 Implementation and Experimental Setup

We implement gang-scheduling inside the Xen hypervisor. While there are many features that can make gang-scheduling more useful in a general-purpose environment, we focus on implementing only a minimal subset as a proof-of-concept. For our experiments, we create two pools of CPUs, one for running the controller domain (Dom0) and the other for gang-scheduled benchmarks. The scheduler only accepts domains with the same number of cores as are in the pool. Gang scheduling is acheived by logically partitioning time into 50ms quantums and assigning each quantum to a domain. This assignment is carried out through a simple round-robin sequence of domains as described in Listing 1.

Xen's scheduling framework will call this `do_schedule` function independently on each CPU when a task's time has run out, or due to hypercalls from guests. Each core will pick the same domain when called during a particular timeslice (since they decide based on the same clock) and run the appropriate virtual CPU, this ensures that all virtual CPUs of a domain run during that domain's timeslice. Each domain will receive $1/n$ of the total processing time, where n is the number of domains.

All experiments are performed on a 2-socket, 28 core test machine (see Table 1 for details) with hyper-threading disabled to eliminate processor variability. Xen domains are created with a minimal paravirtualized Linux 3.14 kernel and run only a single benchmark at a time. When experiments involve multiple instances of a benchmark, we create several domains and start them simultaneously on our gang-scheduled pool of CPUs.

CPU	2× Intel Xeon E5-2697 v3
Clock rate	2.6 GHz (Turbo: 3.6 GHz)
# of Cores	14 per CPU
# of Threads	2 per Core ⇒ 28 per CPU
Memory	64 GiB DDR4-2133
Xen version	4.4.2
Dom0 kernel	3.16.0-25

Table 1: Test machine specifications

4 The Benchmarks

Since we are targeting HPC workloads in this work, we picked two benchmarks from the Department of Energy's Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). This center provides a number of proxy-applications intended to evaluate the performance of advances in supercomputing hardware and software systems. CoEVP is a relatively small, but complex benchmark, while CoHMM is more complicated and longer-running. We also design a microbenchmark to more directly measure the behavior of the different scheduling policies.

4.1 Microbenchmark

```
unsigned long sum = 1;
for(int i=0; i < 30000*NUM_MS; i++)
    sum += pow(i, sum);
```

Listing 2: Microbenchmark code

We have experimentally determined the appropriate values for this loop to take approximately 100ms to run on a single core. Due to the small size of this code, we do not expect that we are measuring any external influences on the running time of the system, such as memory bus access time, etc. We then keep a global count of how many times this code has been run across all the cores. In an attempt to measure the cores becoming out of sync, we track what the count was the last time a given thread completed its run. Assuming that all the cores receive the same amount of execution time, then the count from the global, and the local count of the last time, should differ by no more than the number of threads. In the case that a thread has been determined to be out of sync, we record by how much the thread differs from where it is expected to be, and then sum these values.

4.2 CoEVP

CoEVP implements a simulation of the deformation of a tantalum cylinder fired at a solid wall. It is a scale-bridging technique that simulates many fine-grain models and then interpolates the results into a coarse-grained model. The simulation bridges scale by calculating the deformation in a coarse and fine scale using a Lagrangian finite element model and a fine-scale viscoplasticity model, respectively. The code stores the results of fine-grain simulations in an embedded database that is queried for similar simulations before starting a new one. The fine-grained simulations synchronize internally using OpenMP [9], while the database serves as a synchronization point between simulations.

CoEVP is computationally bound. It makes intensive use of mathematical functions like `pow` and `sqrt`. The current progress of the calculation is periodically output to the console. Other than that, no I/O operations are performed. The workload is parallelized using OpenMP. However, the performance scales well only for a small number of threads as seen in Figure 2.

4.3 CoHMM

CoHMM calculates the shock propagation dynamics in a copper plate that takes an impact at the center. The large-scale deformations are calculated using

```
function do_schedule:
    cur_timeslice = now / QUANTUM;
    end_of_timeslice = (cur_timeslice + 1) * GANG_QUANTUM;
    cur_domain = cur_timeslice % number_of_domains;

    //Amount of time to run the domain for
    ret.time = end_of_timeslice - now;

    //Which virtual CPU to run
    ret.task = domains[cur_domain].v_CPUs[THIS_CORE];
```

Listing 1: Pseudocode for our gang-scheduling algorithm

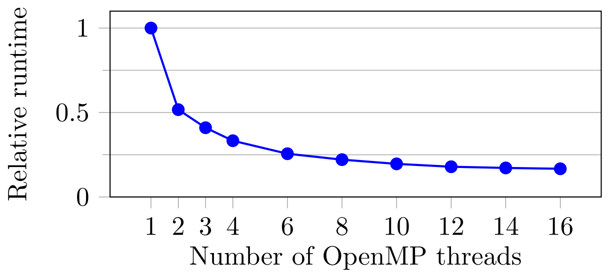

Figure 2: CoEVP scaling characteristic. The runtime is normalized to running it on one core. As it can be seen, the curve flattens out fairly rapidly.

continuum equations. Additionally, fine-scale simulations are performed which span regions of $\sim 10^3$ atoms. This type of modeling is referred to as *heterogeneous multiscale method (HMM)*. The fine-scale computations make up most of the computational cost. Since their results are often very similar, the results are cached in a key-value store, and reused by subsequent computations. This method is called *adaptive sampling*. Entries in the key-value store are interpolated using *Kriging*. CoHMM places the locations of the fine-scale simulations in such a way that reduces synchronization costs between several machines in the cluster that all have their own key-value store and are responsible for calculating a specific region of the total problem. This leads to good parallelism when scaling up the number of machines in the cluster.

In our setup we let the CoHMM instance run on top of a minimal Linux in a single domain with 14 cores. We use Redis as the key-value store which runs in the same domain as a background task. CoHMM communicates with Redis using a loopback network interface. Redis occasionally writes a snapshot to the (RAM) disk. Other than that, no I/O is present. In our measurements we run three iterations of the simulation which results in a runtime of about ten minutes if one domain is running on 14 cores without any other domain running on the same cores. The CPU utilization of the time of the benchmark is depicted in Figure 3. As one can see, CoHMM does not fully saturate all of the CPUs at all times, but does benefit from extra cores during highly parallel phases.

5 Results

We now evaluate each benchmark on our experimental system, starting with the simplest benchmarks and progressing through increasingly complex codes. Results are taken as the average of 8 runs.

5.1 Microbenchmark

Figure 4 graphs the results of running our microbenchmark on multiple 14-core domains. As more concurrent domains are added, Xen's credit scheduler introduces significant variation in the runtimes of each virtual CPU. At 8 domains, each domain misses on average 24000 deadlines out of a total 28000 (86%). This signifies a significant discrepancy in the rate of progress of each virtual CPU in a domain. In addition, there is significant variation between runs with a maximum variation of about 2200 missed deadlines in the 4-domain case. In contrast, the microbenchmark met almost every deadline when run on top of the gang-scheduled CPU pool. In the worst case (at 8 domains) the gang-scheduled pool led to 28 missed deadlines (0.1%). In addition, we observed virtually no variation between runs.

While only a microbenchmark, it is telling to see such high variation between runs on top of the non-gang scheduler. Note also that measurements were taken at a fairly conservative 100ms period. The skew would be even more significant as the inter-barrier

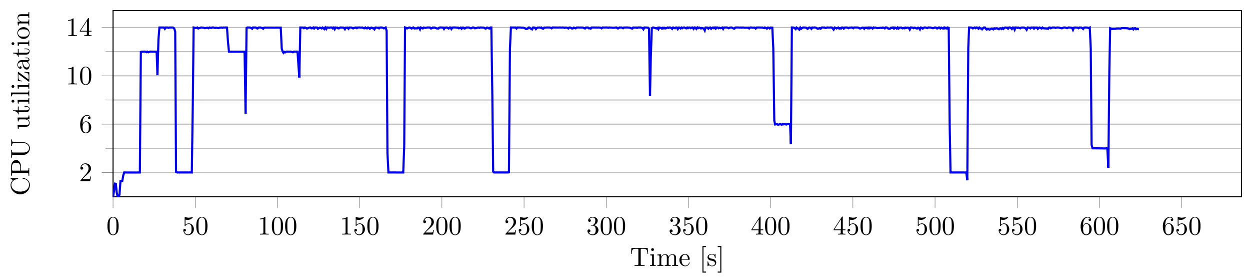

Figure 3: CPU utilization of the CoHMM benchmark over time. A utilization of 14 means that all 14 cores are fully utilized. The mean utilization is 12.37.

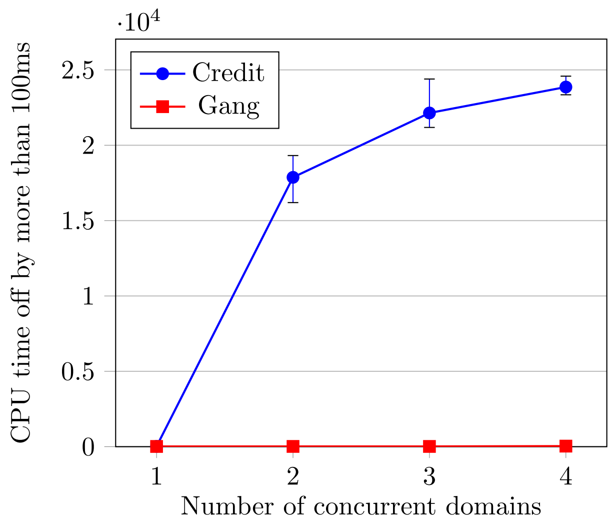

Figure 4: Microbenchmark results over 14 cores while varying the number of concurrent domains.

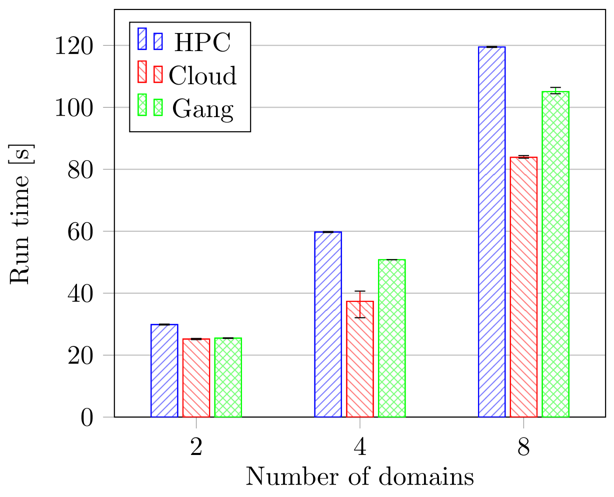

Figure 5: CoEVP benchmark results while varying the number of domains and scheduling policy.

period shrank. Clearly gang-scheduling shows some promise under a fairly contrived set of circumstances.

5.2 CoEVP

Unlike the microbenchmark above, CoEVP is a non-trivial, HPC-oriented benchmark. Figure 5 compares running CoEVP with varying numbers of domains with the different scheduling policies. Each domain was given 14 cores and each run took approximately 13 seconds to complete when run in isolation. In the HPC case, each instance of CoEVP was run in isolation, one after the other. In the cloud (credit scheduler) and gang cases we start all domains simultaneously on a pool of 14 cores.

Running code in batch mode consistently produces the worst results. Since each run takes only 13 seconds, there is a non-trivial overhead of starting and stopping runs that may be adversely affecting the batch mode. With 8 domains running concurrently, the credit scheduler is fastest by far. In the credit scheduler, a blocked VCPU can be swapped out and replaced with a VCPU from another domain that can make progress. This is supported by the increasing margin of improvement credit observes as the number of concurrent domains increases. Since CoEVP does not effectively use all of it's cores (as seen in Figure 6), credit is able to take advantage of the oversubscription by filling gaps in each domains execution with VCPUs from other domains. With only two domains, both gang and the credit scheduler produce similar run times, but as we increase the number of concurrently running domains, gang slows down. Naively, we would expect gang to perform similarly to the batch mode, but even with 8 domains it performs better. While the gang policy cannot run other VCPUs when one is blocked, we hypothesize that each domain in the gang scheduled run is able to service some requests in the background while the other do-

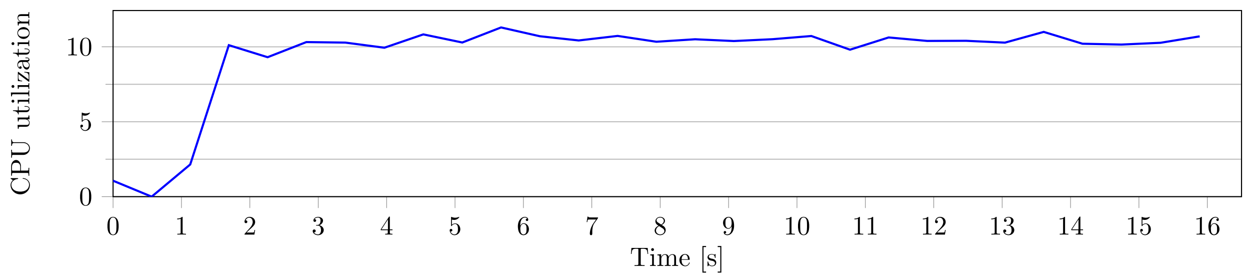

Figure 6: CPU utilization of the CoEVP benchmark over time. A utilization of 10 means that all 10 out of 14 available cores are utilized. The mean utilization is 9.43.

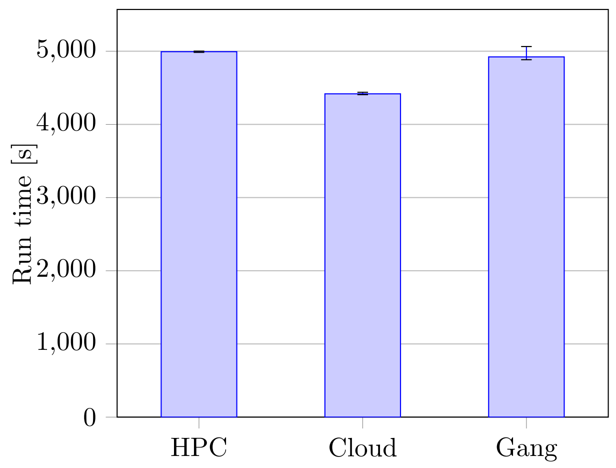

Figure 7: CoHMM run with 8 domains while varying the scheduling policy.

main is running. Although there is little I/O in this benchmark, what does occur may be enough to skew the results on such a short running benchmark.

5.3 CoHMM

CoHMM is significantly more complex than CoEVP and takes considerably longer to run (roughly 10 minutes when run with 14 cores in isolation). The additional run-time should rule out any one-time costs introduced by domain start-up costs or other sources of noise. Figure 7 shows the time taken to complete 8 instances of CoHMM under the three scheduling policies. Like CoEVP, the batch case ran each instance of CoHMM to completion before starting the next while the cloud and gang instances were run simultaneously.

Somewhat unintuitively, the batch case has the worst performance, with gang close behind. While each instance has all of memory and cache to itself, and all threads are scheduled all of the time, overall utilization is poor. From Figure 3 we can see there are significant periods of low utiliztion during the 10 minute run of CoHMM. When the credit scheduler is informed by the guest operating system that a particular VCPU is idle, it is able to pick a VCPU from another instance to run. Both the batch and gang schedulers are bound to schedule every VCPU of a domain, even if that VCPU has nothing to do. Essentially, the credit scheduler is free to fill the "holes" left in each domain with spare work from other domains.

While variability is high under the gang scheduling policy, this variation is due to a single outlier after 8 runs. It is possible that some unfavorable timing of timeslice boundaries has contributed to this outlier, or there could be some as-yet undiscovered bug in our implementation. Without this outlier, the variation in runtimes of the gang scheduler is similar to that of the credit scheduler (0.45%). Without the outlier, the gang-scheduler completed on average 2% faster than the batch scheduler. It is not clear where this performance benefit comes from, perhaps some amount of I/O was able to complete in the background. In either case, gang and batch seem to perform about the same, indicating little to no impact of cache contention or context switch overhead.

6 OS Noise

It is not purely scheduling policy that can lead to lock-holder preemption and barrier latency issues. Scheduling policies were not the primary focus of the work in [11], instead they focused on any source of noise in their system. Researchers at Sandia National Labs spent significant effort to create a predictable, noise-free operating system called Catamount [12]. By carefully re-introducing noise to the system, Riesen et al. were able to measure it's effects. Their work showed that even a 2.5% overhead from kernel noise could induce a 20x degredation in some

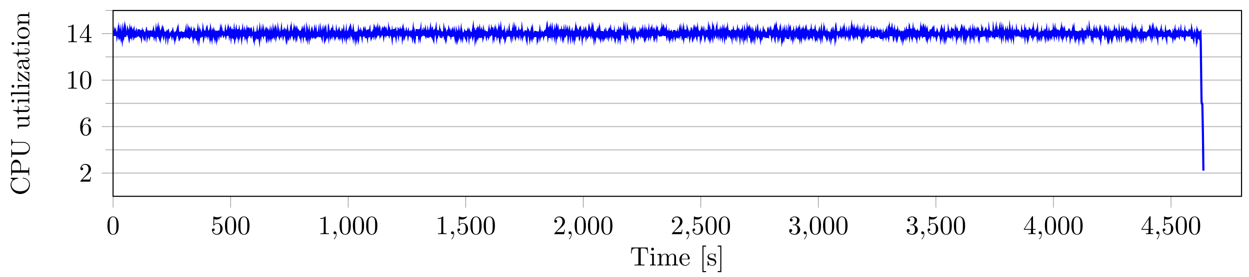

Figure 8: CPU utilization of all 8 domains summed up. The summed utilization shoots over the practical maximum of 14 because the samples were not perfectly synchronized and Xen only provides an approximation to CPU utilization. However, we can see that the combined utilization stays fairly close to the maximum.

benchmarks while other benchmarks experienced virtually no impact. In general, the authors of Catamount went to considerable lengths to remove any source of uncertainty in their system, including scheduler timer interupts, networking events, and virtually all OS background tasks.

6.1 CellOS vs. Linux

Since it was our primary goal to reduce scheduler interference, we initially planned to run our benchmark in the cell using a simple library operating system; CellOS. Doing so would allow us to exactly know the scheduling decisions inside the guest operating system. Another advantage is that there would not be any background tasks running that could interfere with the application.

CellOS is based on GUK [6] which itself is based on Mini-OS [8]. It has been specifically designed to be used on top of the Xen hypervisor. It is written in C and links against newlib [14] which provides the functions of the standard C library. Newlib aims to be a lightweight replacement for glibc and is especially targeted for embedded applications. It is highly portable and is not optimized for specific instruction set architectures. Therefore, operations like the calculation of the square root, that could be carried out using dedicated machine instructions on certain architectures, suffer from performance degradation.

Since our benchmarks are mostly written in C++, the standard C++ library (libstdc++) had to be ported to CellOS. We started by porting CoEVP to CellOS. To avoid porting OpenMP, the parallelized sections were implemented using a simple threadpool. That implementation performed comparably to the OpenMP implementation on Linux. The porting effort for OpenMP would have been significant because in CellOS, only a subset of the pthread library is available. Another stepping stone was the lack of a functioning file system in CellOS, which resulted in many undefined functions when linking against libstdc++.

When running the benchmark in CellOS, several bugs that were caused by race conditions in the CellOS kernel occurred. Debugging was difficult due to the limited functionality of the debugging tools. Once the benchmark was running, we discovered severe performance shortages compared to running the benchmark on Linux. We were able to isolate the math functions implemented by newlib as one of the reasons for those performance issues. The `sqrt` function, for example, is implemented as a Taylor series in newlib while glibc implements it using the x86 instruction `fsqrt`. This results in a large difference in runtime which we could see after replacing the implementation of newlib. Similar effects can be seen at the implementation of `pow`. Newlib's implementation is written in C while glibc implements it using handcrafted assembly. However, `pow` cannot be ported as easily due to dependencies in glibc.

Even after some optimization effort, the performance was still worse by a factor of two. We suspect that besides the math functions, other libc functionality suffers similar performance issues. Since we didn't have the ability to profile CellOS,[1] we can't say with certainty what caused the performance to be so much worse than in the Linux case. We suspect, however, that the CellOS kernel could take up a significant amount of processing time. Another possibility is that Linux puts the processor in different modes than the CellOS does.

After our first experiments, we abandoned CellOS for three major reasons. Firstly, the performance of CellOS was considerably worse than that of Linux. Secondly, we discovered, while running the microbenchmarks, that the Linux scheduler does a good job in keeping the threads synchronized and the

[1]There is a possibility to profile Xen domains (xenoprofile)—even if they run arbitrary operating systems, but newer Linux kernel versions are not supported.

effects of background tasks were not noticeable (see Figure 4). Lastly, deploying new benchmarks, like CoHMM, with Linux was a matter of hours which would have been a significant effort on CellOS.

It is possible that other benchmarks would have experienced significantly different performance, as was observed in [12]. There is one significant difference between our experience and that of [12], that of scale. While they experimented on the 13000 core Cray Red Storm supercomputer, we were using 14 cores of a workstation-class computer.

6.2 Xen Noise

As a bare-metal hypervisor, Xen has many of the same properties as a traditional OS. The Xen scheduler frequently interrupts processors for various purposes, primarily scheduling. The gang scheduling algorithm described in 3 requests that Xen interrupts the CPU once every 50ms. In a perfect environment, this would be the only periodic interrupt and each would be synchronized across all cores. In practice, however, Xen does not seem to perfectly honor these scheduling requests. To observe this, we instrumented Xen to report every invocation of `do_schedule` and recorded both the requested and actual time.

Rather than returning every 50ms as requested, Xen frequently returns much earlier. In many cases it takes up to 3 invocations of `do_schedule` to finish a quantum. Furthermore, Xen does not invoke the scheduler on perfect 50ms boundaries, this leaves a fuzzy edge near quantum boundaries where the gang scheduling guarantees do not hold. In future work we hope to investigate these additional sources of noise and make Xen more predictable.

7 Related Work

The concept of gang-scheduling has been around since at least the 80's. John Ousterhout proposed a method of co-scheduling tasks on the Medusa operating system in 1982 to improve the performance of inter-process communication, specifically RPCs [10]. Ousterhout's algorithm works by statically defining communicating sets of processes and attempting to schedule them simultaneously. Unlike our simplified algorithm, Ousterhout's algorithms will schedule cooperating processes only if they are not blocked, providing better utilization but possibly weaker synchronization guarantees. As mentioned earlier, Lawrence Livermore National Lab implemented gang-scheduling on a number of their supercomputers [5]. While they did see significant improvements while gang scheduling, their experiments focused on machines with $O(1000)$ processors. The performance characteristics of gang scheduling appear to differ significantly with scale. VMware has had some form of co-scheduling in their virtualization products for much of their life [15]. VMware's "relaxed co-scheduling" does not strictly schedule VCPUs simultaneously. Instead, they ensure that the virtual time experienced by two VCPUs from a single domain will never differ by more than some fixed amount of time. By doing this, they bound the maximum time a thread would need to wait for a lock-holder to be re-scheduled. This bounding is usefull for the correctness of guest OS code, but may be insufficient to solve the synchronization issues observed in HPC environments [11]. VMware's scheduler addresses the utlization issues inherent in gang-scheduling by scheduling other domains when a VCPU is blocked, but they continue to count virtual time agains the blocked CPU to mimic the progression of time observed by a non-virtualized CPU.

The issue of lock-holder preemption has been explored by a number of other groups. Researchers at AMD try to detect unusually long waits in kernel mode and then swap out VCPUs that are assumed to be waiting for a spinlock [4]. Waits are detected by modifying the guest kernel's spinlock implementation to signal the VMM when a spinlock has been held longer than is deemed normal. The VMM then tries to schedule a VCPU from the same guest that was preempted in kernel mode. Uhlig et al. simply avoid swapping out VCPUs that are in kernel mode entirely [13]. This approach does not require any modification of the guest OS, but constrains the decision making of the VMM and may be exploitable. Both approaches assume that spinlocks are only held in kernel mode. While this may be true in a traditional operating system, it may not hold as single-application virtual machines and library operating systems blur the line between kernel and user code. As virtual machines become ubiquitous, more customizations are possible in the guest kernel to avoid lock-holder preemption. OSv is an operating system designed from the ground up to be virtualized. It simply avoid spinlocks entirely by using more virtualization-friendly modes of synchronization internally [2].

8 Future Work

While our minimal implementation of gang-scheduling was sufficient to demonstrate scheduling variations and was representative of a uniformly loaded system, it is not very practical. We are developing a full-featured implementation that will allow domains of any size, which should enable us to simulate more realistic environments with complex sources of noise such as the ASCI Q supercomputer described in [11]. Furthermore, as we found, not all applications benefit from gang-scheduling. We hope to explore hybrid systems that allow best-effort domains to be intermingled with more strict gang-scheduled domains. One major source of performance issues with both batch and gang scheduling policies is that these policies are not able to fill the "holes" left by idle virtual CPUs. In the future we would like to explore relaxed co-scheduling policies that are able to utilize these holes while still ensuring that any VCPUs that may be participating in global synchronization are scheduled concurrently.

In general, we are interested in large supercomputer and warehouse-scale computer systems that provide a service-oriented architecture and broad multi-tenancy. We imagine systems such as this to provide virtual machines as the basic unit of allocation, upon which users can deploy complex frameworks such as managed language runtimes or parallel frameworks such as OpenMP. Similar to the concept of an exokernel [3], the hypervisor's role will be to provide a simple and predictable base upon which to build more complex systems. We expect gang-scheduling to be a key aspect of this predictable base.

9 Conclusion

When designing high-performance parallel systems, it is easy to focus on reducing noise in our systems. Our experiments in comparing Linux to more stripped-down library operating systems has shown that even with it's complexity and general-purpose nature, Linux actually introduces very little variation by itself when not oversubscribed. Indeed, our skew-sensitive benchmark observed virtually no skew when run by itself on a single Linux instance. However, when oversubscribed, traditional schedulers such as Xen's credit scheduler can introduce significant skew between processors. Traditional wisdom has held that running applications in batch mode provides the highest performance and lowest variability for parallel workloads. We have shown that gang-scheduling is a technique that can improve the user-experience without sacrificing the properties of a batch processing system. Despite the intuition, however, our results show that significant benefits can be had with a more flexible scheduling policy that allows higher utilization through over-subscription. While there is some evidence that gang scheduling provides benefits to parallel applications with fine-grained synchronization, it has serious draw-backs. The low utiliztion observed on our real-world HPC benchmarks clearly outweighs any performance benefits gained from better synchronization. It would seem that in its simplest incarnation, gang scheduling is not suitable as a general purpose scheduler.

Acknowledgments

We would like to thank Juan Colmenares for his help in implementing the Xen gang scheduler, his insights into the inner workings of Xen were invaluable and our implementation is largely based on his work. We would also like to thank Eric Roman for his help in setting up a test environment, his Unix-foo, and for his insights into the nature of supercomputing workloads. Finally, Steven Hofmeyer's help with CellOS was much-appreciated while porting CoEVP.

References

[1] Adaptive Computing, Inc. *TORQUE Resource Manager.* URL: `http://www.adaptivecomputing.com/products/open-source/torque`.

[2] Cloudius Systems. *OSv: Cloud Operating System.* URL: `http://osv.io`.

[3] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM. `doi:10.1145/224056.224076`.

[4] T. Friebel and S. Biemueller. How to Deal with Lock Holder Preemption, 2008. URL: `http://www.betriebssysteme.org/Aktivitaeten/Treffen/2008-Garching/Programm/docs/Abstract_Friebel.pdf`.

[5] M. A. Jette. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, pages 1–12, New York, NY, USA, 1997. ACM. `doi:10.1145/509593.509647`.

[6] M. Jordan. *GUK.* URL: `https://kenai.com/projects/guestvm/sources/guk/show`.

[7] Linux Foundation. *Xen Project.* URL: http://www.xenproject.org.

[8] Linux Foundation (Xen Project). *Mini-OS.* URL: http://wiki.xen.org/wiki/Mini-OS.

[9] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, July 2013. URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[10] J. Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing Systems*, pages 22–30, Oct. 1982.

[11] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 55–55, Nov 2003. doi:10.1109/SC.2003.10010.

[12] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, 2009. doi:10.1002/cpe.1361.

[13] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 4–4, Berkeley, CA, USA, 2004. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1267242.1267246.

[14] C. Vinschen and J. Johnston. *Newlib: C Library for Embedded Systems.* URL: https://sourceware.org/newlib.

[15] VMware, Inc. *The CPU Scheduler in VMware vSphere 5.1*, 2013. URL: http://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf.

[16] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer Berlin Heidelberg, 2003. doi:10.1007/10968987_3.