2010 IEEE International Conference on Cluster Computing

Designing OS for HPC Applications: Scheduling

Roberto Gioiosa*, Sally A. McKee†, Mateo Valero*‡
*Computer Science Division
Barcelona Supercomputing Center, ES

†Computer Science and Engineering
Chalmers University of Technology, SE

‡Computer Architecture Department
Universitat Politecnica de Catalunya, ES

roberto.gioiosa@bsc.es, mckee@chalmers.se, mateo@ac.upc.edu

***Abstract*—Operating systems have historically been implemented as independent layers between hardware and applications. User programs communicate with the OS through a set of well defined system calls, and do not have direct access to the hardware. The OS, in turn, communicates with the underlying architecture via control registers. Except for these interfaces, the three layers are practically oblivious to each other. While this structure improves portability and transparency, it may not deliver optimal performance. This is especially true for High Performance Computing (HPC) systems, where modern parallel applications and multi-core architectures pose new challenges in terms of performance, power consumption, and system utilization. The hardware, the OS, and the applications can no longer remain isolated, and instead should cooperate to deliver high performance with minimal power consumption.**

In this paper we present our experience with the design and implementation of High Performance Linux (HPL), an operating system designed to optimize the performance of HPC applications running on a state-of-the-art compute cluster. We show how characterizing parallel applications through hardware and software performance counters drives the design of the OS and how including knowledge about the architecture improves performance and efficiency. We perform experiments on a dual-socket IBM POWER6 machine, showing performance improvements and stability (performance variation of 2.11% on average) for NAS, a widely used parallel benchmark suite.

I. Introduction

Operating systems (OSes) have historically been implemented as independent layers between hardware and applications. User programs communicate with the OS through a set of well-defined system calls, and do not have direct access to the hardware[1], which is virtualized by the OS and shared among all user applications. The OS, in turn, communicates with the underlying architecture using control registers specific to each processor. Except for these interfaces, the three layers are practically oblivious of each other. Users generally need not know hardware details, nor should they be required to explicitly port applications to each new platform. This approach provides portability and transparency, but may not provide optimal performance.

Modern Chip Multi-Processors (CMPs) share different levels of internal hardware resources among cores and hardware threads, from the internal pipeline, to the caches, down to the memory controllers. Single thread performance is no longer improving, and applications must exploit more parallelism for overall system throughput. This is especially true for High Performance Computing (HPC) systems where modern parallel applications and CMP architectures pose new challenges for performance, power consumption, and system utilization. HPC applications have long been parallel (usually using MPI programming models), but each HPC cluster node traditionally ran just one process per (single-core) chip. With the introduction of CMP architectures, each cluster node runs several processes per node. Parallel applications have evolved to use a mix of different programming models, such as MPI, OpenMP, UPC, Pthreads, and CUDA. Moreover, CMPs promise to provide the computing power of a cluster within a single node. We argue that in this scenario operating systems should evolve, as well, moving from *scheduling processes* (the classical OS approach) to *scheduling applications*. All processes and threads inside an application should be scheduled as a single entity rather than as a set of individual threads. The OS should find the best distribution (or *thread-to-core binding*) and then "stay out of the way", minimizing OS noise and process migration, which, in turn, have a large impact on performance and scalability.

We present our experience with the design and implementation of the scheduler of High Performance Linux (HPL), a Linux-based operating system designed to optimize performance of HPC applications running on state-of-the-art compute clusters. We show how characterizing parallel applications through hardware and software performance counters drives the design of the HPL. Moreover, we show that including architecture informations improves performance and reduces OS noise.

In the HPC community there are already several examples of customized operating systems, which usually come in one of two flavors: micro-kernels, or customized general-

[1]Some operating systems provide direct access to the hardware (e.g., network devices) to improve performance.

978-0-7695-4220-1/10 $26.00 © 2010 IEEE
DOI 10.1109/CLUSTER.2010.16

purpose OSes. In either case, we argue that the OS should be designed to consider both the needs of the applications and the capabilities of the underlying architecture.

Usually, micro-kernels are lightweight and introduce little OS noise. Their main strength is also their main weakness: the micro-kernel core is designed to be minimalistic and does not provide all the services offered by a monolithic kernel such as Linux. Those services (e.g., memory allocation, virtual file systems, process management) are either not present or are implemented as external components. Communication between the micro-kernel core and such external services is expensive because it involves Inter-Process Communication (IPC) mechanisms. As a result, more and more essential services (e.g., memory management) are moved back into the micro-kernel core. The most successful micro-kernels are those designed for a specific domain, where the services that must be provided by the core are easy to identify. The most obvious example of this approach is the Compute Node Kernel (CNK) [1] developed by IBM for Blue Gene supercomputers. CNK introduces minimal noise and scales to thousands of nodes practically linearly. Finally, micro-kernels support few architectures (if not just one) and are not completely standard (mainly because they result from a single-entity effort): users are usually required to port their applications.

On the other hand, monolithic kernels, in general, and Linux, in particular, provide a large variety of services, support many architectures, and do not require programmer training or application porting. However, they usually introduce considerable OS noise and have limited scalability.

Application portability and ease of programmer training are important for the success of any OS. These activities have to be repeated for every new programmer/machine and have an impact on productivity. To achieve these goals, we customize a general-purpose OS — Linux, in our case — that already provides most services future HPC applications might need and that requires no additional programmer training or application porting. We modify the standard Linux kernel according to the needs of parallel applications and to the characteristics of the underlying architecture in order to maximize performance. We avoid making our solutions architecture-dependent by including only hardware information common to most platforms, like number of cores/threads and cache parameters. The result is a "monolithic" kernel that *behaves* like a micro-kernel. We perform experiments on a dual-socket IBM POWER6 machine and report performance improvements for NAS [2], a widely used parallel benchmark suite. Our results show that we also considerably reduce OS noise and improve performance predictability and peak performance, with no application or libraries modification. Most benchmarks perform within 3.00% variation (2.11%, on average) with respect to their best performances on this machine.

II. HPC Applications

Many scientific, commercial, and military activities require very high computational power. Examples include theoretical physics applications, weather forecasting, physical simulations

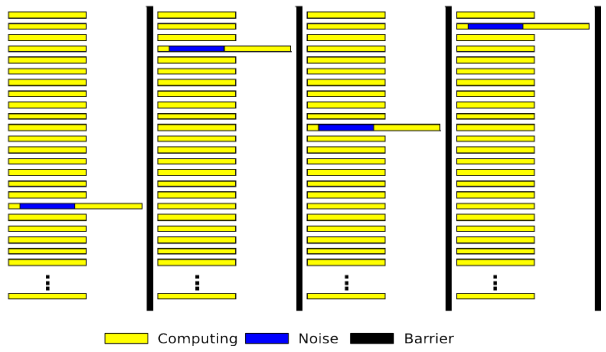


Fig. 1: Effects of process pre-emption on a parallel application

in industrial design, and cryptanalysis, to name a few. Historically, supercomputers were dedicated machines designed to solve specific kinds of computationally challenging problems. The extreme cost of these machines meant that only a few institutions were able to purchase them. Progress in technology has radically changed this scenario. Currently, it is possible to build a supercomputer by means of high-end *Commercial, Off-The-Shelf* (COTS) computers grouped in *clusters*. As a matter of fact, almost all supercomputers built in the last decade consist of HPC clusters, as confirmed by the June 2010 TOP500 Supercomputers list [3]. The OS commonly used in such supercomputers is a version of the Linux kernel.

Most supercomputer applications are *Single Program, Multiple Data* (SPMD) and are usually implemented using MPI [4], [5] or OpenMP [6] (or a combination of both). MPI applications are more common, since shared memory machines with many processors are still very expensive. This situation might change in the near future, for chips with tens or hundreds of cores/threads will likely appear on the market. Nevertheless, since most current applications are based on MPI, we choose to focus on typical behaviors of MPI parallel programs.

MPI applications can be characterized by a cyclic alternation between a *computing phase*, in which each process performs some computation on its local portion of the data set, and a *synchronization phase*, in which processes communicate by exchanging data. These two phases are usually interleaved, and this alternating behavior repeats throughout execution. Good performance requires that all nodes start and terminate each phase at the same time, minimizing time faster nodes waste while waiting for slower nodes to finish a phase.

Achieving this goal is not straightforward, though. An important source of degradation in large MPI applications is operating system *noise*. Essentially, kernel activities delay MPI processes during compute phases. Previous studies measure such OS noise and resulting effects on application execution [7]–[11]. While OS noise in a single node only marginal affects performance (about 1-2%, on average), its effects in a large-scale cluster may become dramatic. When scaling to thousands of nodes, the probability that in each computing phase at least one node is slowed by some long kernel activity approaches 1.0. This phenomenon is *noise resonance* [7], [9].

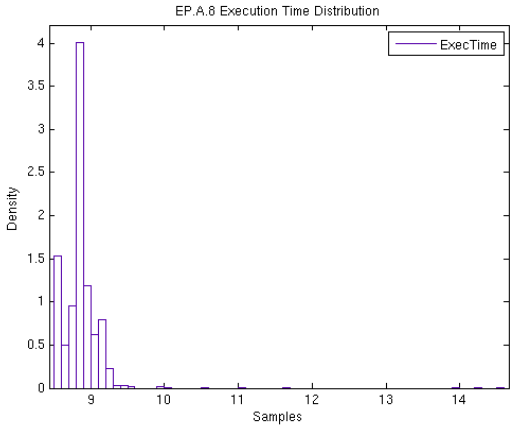


Fig. 2: Execution time distribution for NAS *ep.A.8*

As a practical example, consider the Linux kernel, which tries to be fair to all runnable processes/threads, whether part of a parallel application or not. The OS may occasionally suspends a parallel application thread in order to run a lower priority thread (e.g., statistics collectors or kernel threads). The suspended thread reaches the synchronization point late, and the entire parallel application may suffers performance degradation. Figure 1 depicts this scenario.

III. Running HPC applications in Linux

In order to understand the problems an OS like Linux introduces in HPC applications, we analyze the NAS [2] suite, which contains some of the most common HPC benchmarks. As the number of available cores increases, scalability becomes crucial to good performance. Our primary goal thus becomes understanding the main OS components that reduce scalability. The secondary goal is understanding limits to performance, i.e., why Linux performs worse than micro-kernels. The two goals are coupled: any activity not related to a HPC application is considered overhead that limits performance and can happen asynchronously, introducing performance variability. Performance degradation and variation affect scalability if their impact increases with the number of cores. Performance variability is an index of OS overhead and must be reduced as much as possible for good scalability and overall performance. To evaluate OS effects on scalability, we need parallel applications with good speedup. According to Amdahl's law [12], application speedup is limited by the amount of time spent in synchronization, hence applications that spend less time synchronizing scale better. Among all the NAS benchmarks, *ep* enjoys the least synchronization overhead because it is embarrassing parallel. We executed a parallel MPI version of *ep.A.8* (eight MPI processes, class A inputs) 1000 times and analyzed execution time variation. We used class A input even though it is the smallest data set for the NAS applications because it highlights the effects of OS noise: even minimal OS activity has a visible impact on the application's performance when the execution time is small. Such schemes for measuring OS noise are common in other research [7], [13], [14]. Section V provides more representative experiments with larger data sets.

Ideally, the execution time of an application (especially on an HPC system) should be as low as possible and should be constant (no variation). However, Figure 2 shows that execution times for *ep.A.8* are not constant. Run times vary considerably — from 8.54 to 14.59 seconds. Since *ep.A* is an embarrassing parallel application, another way to look at this experiment is to consider the 1000 iterations of *ep.A* running eight MPI tasks as a single iteration running 8000 tasks (a common configuration for current supercomputer applications). In this case, the entire application (8000 tasks) would take 14.59 seconds to complete, even though most of the MPI tasks would reach their barriers after 8.54 seconds.

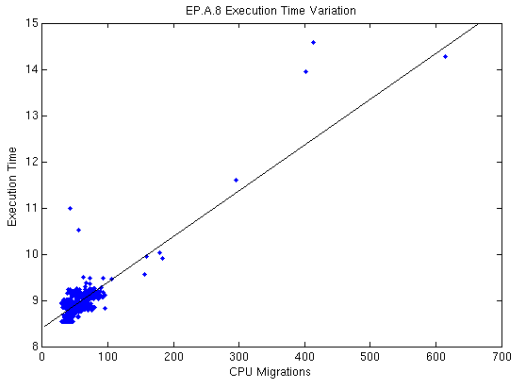
In order to understand why the performance of *ep.A.8* varies by up to 1.71×, we use the performance monitoring tool (*perf*) introduced in Linux 2.6.31. The latest Linux kernels, in fact, provide new support for accessing hardware performance events and exports interesting OS activity statistics in the form of software performance events. We correlate information obtained from software performance events with the performance variation of *ep.A.8* to reveal an empirical relationship between performance degradation and OS scheduler activity. Figures 3a and 3b show that execution time increases with the number of CPU migrations and the number of context switches. These are not the only factors affecting performance variation, although both clearly introduce overhead. In Section V, we show that reducing CPU migration and process preemption indeed reduces variation, confirming the relationship suggested by Figures 3a and 3b. Different memory layouts could also result in performance variation. In our experiments, however, we notice no measurable performance variation after mitigating CPU migration and process preemption.

According to these results, the scheduler is one of the main components that introduces overhead and performance variability. In this paper we focus on two sources of OS noise related to the task scheduler:

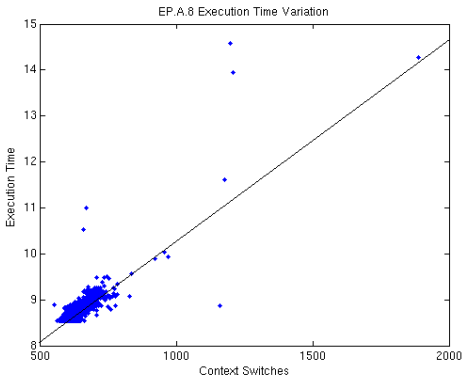
Process preemption. The OS scheduler may decide to run another process in place of an HPC task. If this happens, the suspended process will arrive at the synchronization point later, possibly delaying all the other HPC tasks. Figure 1 shows this case for an application that runs several iterations and synchronizes the processes through barriers or all-to-all reduces.

CPU migration. The scheduler may decide to move a process from one CPU to another. This may happen for load balancing [15] or to take advantage of an idle CPU. The Linux load balancer does not distinguish between the parallel application and the rest of the user and kernel daemons and balances the load assigning (roughly) the same number of runnable tasks to each core.

Both preemption and CPU migration have *direct* and *indirect* effects on application performance. Suspending a process



(a) CPU Migrations



(b) Context Switches

Fig. 3: Software performance events for *ep.A.8*

introduces direct overhead because it makes no progress when not running. Moreover, a non-HPC process may evict some of the HPC task's cache lines, causing extra misses when the HPC task restarts. Furthermore, when the OS moves a task to another CPU, that task may lose its cache contents[2] and cannot run at full speed until the cache rewarms.

IV. HPL

HPL is Linux-based kernel for HPC systems that solves some of the problems of standard Linux kernels in terms of overhead and performance variability to improve the performance and scalability of HPC applications. In this paper we focus on the design of a new task scheduler, since the scheduler appears to be a main OS noise factor affecting performance variation. Linux kernel 2.6.23 introduces a process task scheduler named *Completely Fair Scheduler* (CFS), together with a *scheduler framework*. The scheduler framework, in turn, is divided in two main components: a set of three *Scheduling Classes* and a *Scheduler Core*. The Scheduling Classes are *objects* that provide suitable methods for any low-level operation invoked by the Scheduler Core (e.g., selecting the next task to run). Each Scheduling Class contains one or more policies.

Each CPU has a list of Scheduling Classes. Each class, in turn, contains a list of runnable processes belonging to a class policies. The first class (the highest priority) contains real-time processes; the second (the new CFS class) contains normal processes; and the last class contains the idle process. The ordering of the Scheduling Classes introduces an implicit level of prioritization: no processes from a lower priority class will be selected as long as there are available processes in a higher priority class. It follows that the idle process is never selected if there are runnable processes in other classes.

When the scheduler is invoked, the Scheduler Core looks for the best process to run from the highest priority class (i.e., the Real-Time class). If the class is empty, i.e., no runnable process is available, the Scheduler Core moves to the next class. This operation repeats until the Scheduler Core finds a runnable task to run on the CPU. Notice that the idle class always contains at least the idle process, thus the scheduler's search cannot fail.

By taking advantage of the new scheduler framework, we implemented the HPL task scheduler as a new Scheduler Class between the standard Real-Time and CFS Linux classes. A user can move an application to the HPC class by means of the standard `sched_setscheduler()` system call or via our modified version of `chrt`, which provides support for our new Scheduling Class.

As we show in Section V, HPL reduces the overhead caused by process preemption because it will not select any process from the CFS class (to which user and kernel daemons belong) if there are HPC tasks available. For the same reason, the scheduler reduces the number of expensive context switches (that may evict data from the cache).

Since HPC systems usually run at most one task per core or hardware thread, we expect to have one process in the HPC class of every CPU (maybe two or three in special cases such as initialization and finalization). A complex algorithm to select the next task to run is not warranted. We thus opt for a simple round-robin run queue.

In the new scheduler framework, load balancing, i.e., evenly splitting the workload among all the available processor domains [15] at core-, chip- and system-level, is also performed within each Scheduling Class. Every Scheduling Class implements a workload balancing algorithm, which means each CPU has roughly the same number of real-time or normal tasks. The workload balancer is invoked whenever the kernel detects a significant imbalance or if one processor is idle. In the latter case, the idle CPU tries to pull tasks from other run-queue lists. HPL performs load balancing only when there is an opportunity for performance improvement, like correcting application load imbalance. We do not address this problem here, but our OS solution for balancing HPC applications through smart hardware resource allocation is detailed elsewhere [16], [17].

[2]This overhead is mitigated if the source and destination cores share some levels of cache.

Other reasons to perform load balancing include power consumption, temperature issues, and cache interference. In Section V, below, load balancing occurs only when launching an application. Balancing tasks dynamically simply introduces too much OS noise via both direct and indirect overheads.

The HPL load balancing algorithm depends on the underlying processor architecture topology and its characteristics. Our test processor (IBM POWER6) does not share cache levels between cores, so performance improvement through load balancing is limited: we find that induced overheads exceeds benefits. HPL thus performs load balancing only when a `fork()` is executed. We leave power and temperature balancing for future work.

The HPL load balancer is optimized for performance: we consider the architecture topology (how many hardware threads per core, how many cores per chip, cache sharing, etc.) when performing load balancing. For example, an IBM POWER6 processor consists of two cores per chip, and each core, in turn, consists of two hardware threads. The two cores per chip share neither the L1 nor L2 cache[3]. To optimize performance, our load balancer tries to use all available cores by assigning one process per core when the number of HPC tasks is less than or equal to the number of cores. When the number of HPC processes is higher than the number of cores, the scheduler uses the second hardware thread of each core.

Linux provides the user with several scheduler parameters that may be useful to improve performance of HPC applications and may suggest that a new scheduler is, in fact, not required: `nice`, Real-Time scheduler and CPU affinity.

At every `tick`, the scheduler updates the *dynamic priority* of the running process on a given CPU. The dynamic priority of a process is the sum of its *static priority* (which can be increased/decreased with the `nice()` system call) and other run-time factors. The dynamic priority decreases while the process runs, and when the process exceeds its time quantum, its dynamic priority becomes zero. On the other hand, the dynamic priority increases while a process sleeps, so that when the task again becomes runnable its probability of obtaining a CPU is high[4]. Moreover, the Linux scheduler tries to minimize CPU migrations and to preserve cache contents by assigning a bonus to processes that have been recently running on the current CPU. If, at any time, a process with higher dynamic priority than the running task becomes available (i.e., runnable but not running) on any CPU, the running task is preempted. Note that the kernel also tries to be fair with all runnable processes, ensuring that no task starves waiting too long for a CPU. Since the dynamic priority of a process decreases as it runs, sooner or later processes will migrate (forced by task balancing or idle CPUs) and running tasks will be preempted. It follows that a higher static priority does not guarantee that a process will not be preempted: since process preemption depends on the dynamic priority, a user daemon that has been sleeping for enough time or with high priority (e.g., the `migration` kernel daemon) can preempt a process with a high static priority. This should not be considered a Linux design flaw: Linux was not designed for HPC, but as a general purpose OS and most of the assumptions about and needs of HPC systems do not hold in general.

[3]Some POWER6 chip blades have an external, shared L3 cache, but our dual-socket blade lacks this.

[4]This approach favors interactive tasks over batch tasks

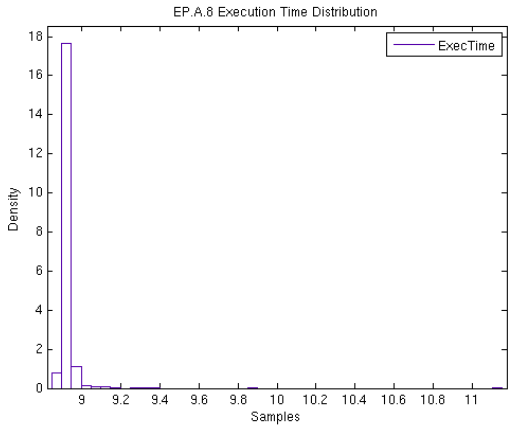


Fig. 4: Execution time distribution for NAS *ep.A.8* with RT scheduler

Instead of increasing the `nice` value, an HPC application could run with the RT Scheduler (`TASK_FIFO` or `TASK_RR`). In this case, HPC tasks will have higher priority than normal CFS processes: regardless of their dynamic priorities, the scheduler always selects RT tasks over normal tasks. Figure 4 shows the execution time distribution of *ep.A.8* running with the RT scheduler. As the figure shows, the RT scheduler provides more stability, but does not solve the problem: *ep.A.8* execution time is more constant with the RT scheduler compared to the standard scheduler, but there is still variation (maximum observed execution time is 11.14 seconds, which corresponds to 208 CPU migrations and 1444 context switches). Process preemption is reduced — but not eliminated — and load balancing can still force process migration because the kernel daemon `migration` has high RT priority.

Though it may seem counter-intuitive, load balancing is a bigger problem for the Real-Time scheduler than for the CFS scheduler. In fact, both load balancing algorithms are based on the number of runnable tasks in the run queues (either Real-Time or CFS), but since there are fewer real-time tasks than CFS tasks, the probability of triggering a load balancing operation is higher with the Real-Time scheduler. In our configuration, for example, there are three domain levels: chip, core, and hardware thread. If, at some point, two HPC application tasks running on the same core are not runnable (e.g., if they are waiting for a message), the corresponding idle CPUs will immediately try to pull some real-time tasks from busier CPUs. Since each CPU runs one real-time task, i.e., each is busy at the same level, the idle processor may pull a task from any busy CPU, triggering any sort of task migration. Moreover, if the number of real-time tasks is odd

or does not map well to the scheduling domain, every attempt to balance the load will result in an imbalanced configuration, triggering yet more load balancing operations. HPL uses a similar solution and favors HPC applications over other low-priority processes, which reduces process preemption, but, a the same time, performs no load balancing unless required for performance improvement.

A common solution for avoiding load balancing is to bind a task to a specific CPU by means of the `sched_setaffinity()` system call. The OS is then forced to run each task on the assigned CPU. This does not mean that the OS will not attempt to balance tasks, but that every attempt to move a task from one CPU to another will fail. Moreover, since every attempt to balance the load fails, the OS will attempt load balancing at every tick, introducing direct overhead. Static solutions introduce low run-time overhead but require the user to set the correct thread-to-core binding before running the application. This task is not trivial and must be repeated for each application, input set, and architecture. Several off-line solutions [18] have been proposed to automatically find an optimal thread-to-core binding through application profiling, but these still suffer the problem inherent to every static solution: the binding is always the same, even when new run-time conditions require a change in configuration. Finally, static thread-to-core binding solutions do not allow CPU migration when it is actually useful (e.g., load imbalance [16], [17] or power [19]).

To overcome the drawbacks of static solutions, sophisticated run-time systems [20] may attempt to dynamically change thread-to-core bindings accordingly to run-time conditions. Most such solutions introduce a user-level scheduler for HPC applications that may add considerable run-time overheads caused by repeated system call invocations (e.g., `sched_setaffinity()`) and from OS attempts to perform scheduling independently from the user-level scheduler. In contrast, HPL performs scheduling and load balancing at kernel level without system call context-switches and without performing scheduling twice (once at user level and again at kernel level) resulting in lower overhead and better scalability for HPC applications.

V. Experimental Results

We test HPL on an IBM *js22* server equipped with two IBM POWER6 chips and 16 GB of memory. IBM POWER6 processors contain two cores per chip, and each core contains two hardware threads, for a total of eight hardware threads[5]. Our current HPL implementation is based on Linux version 2.6.34. We compare our results to those from experiments with an unmodified Linux kernel. We report results for the MPI version of the NAS Parallel Benchmarks [2] version 3.3 compiled with GCC 4.3. Unless otherwise stated, we report statistics over 1000 executions of each benchmark.

[5]Some of the NAS benchmarks cannot be executed with eight tasks. For example, *bt* requires a number of tasks that is a perfect square. We omit such benchmarks from our results.

Figures 3a and 3b in Section IV show that execution times of *ep* increase with the number of CPU migrations and context switches, respectively. HPL aims to reduce scheduler overhead (and thus OS noise) by reducing CPU migrations and avoiding process preemption for HPC applications. We use the new Linux performance tool (*perf*) to count CPU migrations and context switches during parallel application execution.

Applications require different kernel services (system calls) that cause user-to-kernel mode switches. These modes switches do not cause context switches (from one process to another) and are necessary for correct application behavior (for example, to perform memory allocation, create reports, or send messages) and should be considered part of an application's execution. Other context switches, such as those introduced by the OS when the scheduler preempts an HPC task to run another process, are not part of normal execution and instead appear to the application as OS noise. HPL reduces the latter context switches by not allowing a normal process to be selected to run on a CPU as long as there is at least a runnable HPC task in that CPU's run-queue list. If there are no runnable HPC tasks, HPL looks for runnable tasks in the other scheduling classes (eventually selecting the idle task). Running a normal process may introduce indirect overhead (the are no HPC tasks running so there is no direct overhead), but jobs such as low priority system activities (statistics collection or cluster management) must nonetheless execute. Instead of interrupting HPC tasks asynchronously during their execution, HPL performs such activities when there are no HPC tasks running on a CPU.

Table Ia shows that there are several context switches during parallel application execution, even when using a small data set. *ep* has, on average, 652.62 context switches in class A and 1333.70 in class B. *ep* is an embarrassing parallel application (almost no communication), and increasing the data-set size only increases the computation part of the application, without generating more communication or system calls. It follows, that he extra 681.08 context switches for the class B data set are caused by the OS.

Table Ia also reports the number of CPU migrations performed by the standard Linux kernel. CPU migrations are either voluntary (through the `sched_setaffinity()` system call) or caused by load balancing. Since the NAS benchmarks do not invoke `sched_setaffinity()`, we conclude that all CPU migrations in Table Ia are initiated by the OS. The standard Linux kernel performs load balancing in two cases: 1) when a CPU is idle, and 2) when the system load is unbalanced and the balancing timer has expired. In either case, load balancing introduces both direct (scheduler balancing) and indirect (cache affinity) overhead. In these experiments, HPL performs load balancing only when the application starts and accordingly to the system topology. In our test system, HPL first balances the load between the two chips, then between the cores in a chip, and finally between the hardware threads within a core. Once the HPC application is running, HPL tries to stay "out of the way", minimizing OS noise. Nonetheless, avoiding load balancing for the HPC applications is insufficient. In fact, the

TABLE I: Scheduler OS noise for NAS

(a) Standard case

Bench	CPU Migrations			Context Switches		
	Min.	Avg.	Max.	Min.	Avg.	Max.
cg.A.8	30	63.61	2078	460	602.57	5755
cg.B.8	28	90.62	3499	1726	2011.80	8243
ep.A.8	29	52.41	615	550	652.62	1886
ep.B.8	28	69.02	2536	1198	1333.70	5239
ft.A.8	20	53.02	565	318	575.10	1609
ft.B.8	28	51.23	1163	1111	1222.50	3258
is.A.8	29	52.18	160	396	537.35	956
is.B.8	28	52.88	370	519	610.93	1213
lu.A.8	18	70.79	1368	219	1030.40	3870
lu.b.8	29	69.04	3657	2518	2933.50	9131
mg.A.8	29	54.73	590	91	556.24	1776
mg.B.8	29	54.68	853	531	660.43	2396

(b) HPL case

Bench	CPU Migrations			Context Switches		
	Min.	Avg.	Max.	Min.	Avg.	Max.
cg.A.8	10	11.52	14	333	356.32	391
cg.B.8	10	12.31	21	343	374.72	484
ep.A.8	10	12.02	18	315	344.77	436
ep.B.8	10	12.04	19	329	365.39	472
ft.A.8	10	11.43	17	331	361.32	413
ft.B.8	10	12.11	19	337	365.29	414
is.A.8	10	11.39	14	326	347.37	382
is.B.8	10	12.07	23	340	354.97	374
lu.A.8	10	12.84	21	325	361.81	604
lu.b.8	10	12.97	22	340	381.46	455
mg.A.8	10	11.94	22	357	386.60	423
mg.B.8	10	12.55	17	357	386.44	422

kernel attempts to balance CFS tasks, which introduces some OS noise because these tasks have no chance to run when there are runnable HPC tasks. This overhead is less than the standard load balancing overhead because there are no CPU migrations (thus no indirect overhead), but it is nonetheless undesirable. Therefore, HPL performs no load balancing for *any* scheduling class in order to reduce direct overhead along with indirect overhead.

Table Ib shows how HPL drastically reduces the number of CPU migrations. Essentially, HPL only performs 10 CPU migrations. During the initialization there is one migration for each MPI task as it is created (for a total of eight migrations); one migration occurs when `mpiexec`[6] is created; finally, one migration is caused by `chrt` when `mpiexec` returns control, and at least one is created by the `perf` Linux tool we use to report performance counter values. Note that both `chrt` and `perf` run either before the HPC application starts or once it has completed. They belong to the CFS class, and HPL does not prevent load balancing for such tasks if there are no runnable HPC tasks. `chrt` and `perf` CPU migrations occur when the HPC application has completed but the `perf` tool still reports them because we perform system-wide performance counter monitoring.

Table Ib also shows that the number of context switches has been consistently reduced and does not vary with data-set size. On average, *ep* has 344.77 context switches for class A and 365.39 for class B[7].

Table II shows the most important results of our experiments: the impact of HPL on the application's execution time and performance variability. The table shows that all NAS benchmarks perform better with HPL than with the standard Linux kernel and that performance variations have been considerably diminished. For example, *ep* executes with less than 1% performance variation[8] across all repetitions, which indicates that OS noise has been reduced and that the OS can run HPC applications very effectively. The other NAS benchmarks experience less than 3% performance variation (except *lu.B.8*). In this paper, we focus on the scheduler design of HPL and do not address micro-noise [7], [10] from the local timer interrupt. We deliberately present experiments in this way in order to isolate the impact of our scheduler. HPL uses NETTICK [21] to reduce periodic timer interrupts and, consequently, to reduce micro-noise. We believe that peak performance can still be improved by using NETTICK and that HPL can operate even more like a micro-kernel.

We conduct our experiments on HPL without changing any application source code, any application binaries, or any libraries, and we run on a standard cluster node with no special OS hardware support. Nevertheless, we achieve good performance stability for the NAS benchmarks studied here.

VI. Related Work

Operating system noise and its impact on parallel applications have been extensively studied via various techniques, including noise injection [14]. In the HPC community, this problem was first demonstrated by Petrini et al. [9]: they explained how OS noise and other system activities (not necessarily at the OS level) dramatically impact the performance of a large cluster. The impact of the operating system on classical MPI operations, such as collective, is examined in Beckman et al. [22]. Petrini et al. [9] observed that the impact of the system noise when scaling on 8K processors was so large (because of *noise resonance*) that leaving one processor idle to take care of the system activities led to a performance improvement of 1.87×. Though Petrini et al. [9] did not

[6]We use a modified version of `chrt` that implements our new scheduling class, and we pass `mpiexec` as an argument. Hence, `mpiexec` also belongs to the HPC Class. This introduces no run-time overhead because `mpiexec` only forks the other MPI tasks and waits for them to finish.

[7]The small variation is caused by nondeterministic behavior in `perf` and `chrt`.

[8]Here variation is computed as the difference between maximum and minimum performance values divided by the minimum value.

TABLE II: NAS Execution Time: Std. Linux VS HPL (seconds)

Bench	Std. Linux				HPL			
	Min.	Avg.	Max.	Var. %	Min.	Avg.	Max.	Var. %
cg.A.8	0.69	1.04	46.69	6608.70	0.68	0.69	0.70	2.94
cg.B.8	36.98	42.04	126.48	242.02	36.96	37.27	38.17	3.27
ep.A.8	8.54	8.87	14.59	70.84	8.54	8.55	8.57	0.35
ep.B.8	34.14	34.69	53.34	56.24	34.14	34.19	34.33	0.56
ft.A.8	2.27	2.50	9.07	327.31	2.05	2.06	2.08	1.46
ft.B.8	22.56	22.91	41.78	85.20	22.58	22.66	22.71	0.58
is.A.8	0.35	0.57	3.27	832.29	0.35	0.36	0.36	2.86
is.B.8	1.82	1.88	4.82	164.84	1.82	1.83	1.84	1.10
lu.A.8	17.56	19.34	50.85	189.58	17.71	17.79	18.00	1.64
lu.B.8	71.93	79.37	140.03	94.68	71.81	73.51	77.64	8.12
mg.A.8	1.40	1.60	7.80	457.14	0.96	0.97	0.97	1.04
mg.B.8	4.48	4.96	28.35	532.81	4.48	4.93	4.54	1.34

identify every source of OS noise, a following paper [7] identified timer interrupts (and the activities started by the paired interrupt handler) as the main source of OS noise (*micro OS noise*). Others [10], [11] found the same result using different methodologies: timer interrupts represent 63% of the OS noise. The other major cause of OS noise is the scheduler, i.e., what we address here: the local kernel can swap HPC processes out in order to run other (lower priority) processes, including kernel daemons. This problem has also been extensively studied [7], [9]–[11], [23], and several solutions are available [24]. All the studies confirm that OS noise can be classified in two categories: high-frequency, short-duration noise (e.g., timer interrupts) and low-frequency, long-duration noise (e.g., kernel threads) [14]. Impact on HPC applications is higher when the OS noise resonates with the application, i.e., high-frequency, fine-grained noise affects more fine-grained applications, and low-frequency, coarse-grained noise affects more coarse-grained applications [9], [14].

The literature provides several examples of OSes specifically designed to obtain maximum performance for HPC applications, the most remarkable example being IBM CNK for Blue Gene machines. Current solutions can be divided into three classes: micro kernels, lightweight kernels (LWKs), and monolithic kernels. *L4* is a family of micro-kernels originally implemented as highly tuned Intel i386-specific assembly language code [25]. L4 was later extended in various directions, both to achieve a higher grade of platform independence and also to improve security, isolation, and robustness. *Exokernel* [26]–[28] was developed by MIT in order to provide greater freedom to developers in terms of hardware abstractions decisions. The main idea behind MIT exokernel is that the OS should act as an executive for small programs provided by the application software. The exokernel essentially only guarantees that these programs use the hardware safely. *K42* [29] is a high performance, open source, general purpose research operating system kernel for cache-coherent multiprocessors developed by IBM. The OS was specifically designed for multiprocessor systems exploiting both spatial and temporal locality in code and data, which results in substantial performance advantages. K42 was designed to scale to hundreds of processors and to address the reliability and fault-tolerance requirements of large commercial applications. One of the key ideas of K42 is the ability to plug in building-blocks to allow applications to customize and optimize OS services.

Sandia National Laboratories has a history of developing LWKs dating back to the early 90s. Riesen et al. [30] described the evolution, design and performance of Sandia's LWKs, highlighting the benefits of their implementations over standard OSes like Linux or OSF/1. As we do here, the authors identified predictable performance and scalability as major goals for HPC systems. One important distinction between Catamount (the last version of Sandia Lab's LWK) and HPL is in the scheduler implementation: Catamount provides a privileged user-level process (PCT) that performs scheduling functions while HPL performs scheduling in kernel mode. Catamount's approach has the advantage that users can write their own specific scheduler. On the other hand, communication between the lowest-level component (Quintessential Kernel) and the PCT may introduce overhead. In our experience, few users are willing to write schedulers specific to their applications; most prefer portability (notice that Catamount provides a default scheduler for these users). We thus choose to minimize performance overhead in this case.

IBM *Compute Node Kernel* for Blue Gene supercomputers [**?**], [31] is a very good example of optimized OS for HPC systems. CNK is a standard, vendor-supported software stack that provides maximum performance and scales up to thousands of nodes practically linearly. CNK is neither a micro-kernel nor a monolithic kernel: the OS is a Lightweight kernel that provides only the services required by HPC applications in the most efficient way. Micro-kernels usually partner with library services in the OS to perform traditional system functionalities. CNK, on the other hand, maps hardware to user and does not perform all system functionality. The disadvantages of

CNK come from its highly specialized implementation: the OS can run a limited number of processes/threads, it provides no `fork`/`exec` support, and it only runs dedicated Blue Gene binaries. Moreira et al. [31] show two cases (socket I/O and support for Multiple Program Multiple Data (MPMD) applications) where IBM had to extend CNK to satisfied the requirements of a particular customer. These situations are quite common with micro and lightweight kernels.

There are several variants of Linux among the customized general-purpose OSes. ZeptoOS [32]–[34] is an alternative, Linux-based software stack available only for Blue Gene machines. The ZeptoOS kernel offers several of the same advantages (it is easy to tweak and debug, runs arbitrary jobs, and supports interactive login to compute nodes, e.g.) and disadvantages (the OS is not as fast as CNK) of HPL.

Jones et al. [23] provides a detailed explanation of the modification introduced to IBM AIX to reduce the execution time of MPI `Allreduce`. The authors noticed that some of the OS activities were easy to remove while others required AIX modifications. The authors prioritize HPC tasks over user and kernel daemons by playing with the process priority and alternating periods of favored and unfavored priority. HPL also prioritizes HPC tasks, but rather than going through all tasks and changing priorities, the OS prioritizes a *class* of tasks (the HPC class over the CFS class). This makes it easier to favor critical system daemons (as the I/O daemon found by Jones et al. [23] when running ALE3D): simply reducing the HPC scheduling class priority would allow daemons to run.

An interesting comparison between CNK and Linux on BlueGene/L is proposed by Shmueli et al. [35]. The authors conclude that one of the main reasons limiting Linux scalability on BlueGene/L is the high number of TLB misses. Although the authors do not address scheduling problems, they achieve a scalability comparable to CNK by using the HugeTLB library (although not with the same performance). We plan to follow the same technique with HPL in future work.

Mann and Mittal [13] use the secondary hardware thread of IBM POWER5 and POWER6 processors to handle OS noise introduced by Linux. They show OS noise reduction mainly using Linux features (e.g., Real time scheduler, process priority, and interrupt redirection). As we show in Section III, these features are not enough for current HPC systems, and, in fact, the authors cannot completely eliminate OS noise. Moreover, the OS noise reduction comes at the cost of losing the computing power of the second hardware thread. Interestingly, Mann and Mittal consider SMT interference a source of OS noise, but we consider SMT interference a source of hardware interaction beyond the scope of this work.

VII. Conclusions and Future Work

Future operating systems should be designed to satisfy the requirements of parallel applications, which are entirely different from those of sequential applications or multi-programmed workloads. Moreover, the operating system should interact with the underlying processor to exploit modern Chip-Multi core or Multi-Thread processors. The hardware, the operating system, and the applications can no longer remain isolated, and instead should communicate and cooperate to achieve high performance with the minimal power consumption.

In this paper we presented our experience with the design of the scheduler of HPL, a Linux-based operating system specifically designed to improve HPC applications' performance by reducing OS noise and, therefore, performance variability. HPL has been designed to solve some of the problem HPC applications may incur to when running on a cluster node and that can limit their scalability. To understand the OS bottlenecks we profiled HPC applications running on a state-of-the-art machine using the new Linux performance events infrastructure. According to our results, context-switches and, especially, CPU migrations have a direct relationship with the applications' performance degradation and variability. The HPL scheduler effectively solves these problems: no low priority process can preempt an HPC tasks or be selected to run on a CPU as long as there are runnable HPC processes. Moreover, load balancing is performed accordingly to the processor's topology and characteristics (number of chip, cores per chip, hardware threads per core, shared cache levels) and only when it is actually required. On our test machine, equipped with two IBM POWER6 processors, HPL only performs load balancing for HPC tasks when a `fork()` is invoked and avoid CFS tasks load balancing when the parallel application is running. Our results, performed with an MPI version of the NAS parallel benchmark suite, show that the number of context-switches and CPU migrations have effectively been diminished to the minimum. These results have a direct impact on performance variability: NAS benchmarks perform within 3% of their average execution time, two order of magnitudes better than the standard Linux kernel.

We will extend HPL taking into account the power dimension and TLB performance. We would also like to port HPL to Blue Gene compute nodes, which would allow us to compare HPL to CNK and ZeptoOS.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN-2007-60625 and JCI-2008-3688, as well as the HiPEAC Network of Excellence (IST-004408). The authors thank all the anonymous reviewers and the shepherd for constructive comments and suggestions. The authors also thank Robert Wisniewski from IBM T.J. Watson research center and Marco Cesati from University of Rome "Tor Vergata" for their technical support.

References

[1] M. E. Giampapa, R. Bellofatto, M. A. Blumrich, D. Chen, M. B. Dombrowa, A. Gara, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, B. J. Nathanson, B. D. Steinmacher-Burow, M. Ohmacht, V. Salapura, and P. Vranas, "Blue Gene/L advanced diagnostics environment," *IBM Journal for Research and Development*, 2005.

[2] NASA, "NAS parallel benchmarks," http://www.nas.nasa.gov/Resources/Software/npb.html.

[3] "TOP500 Supercomputer Sites list," http://www.top500.org/.

[4] M. Forum, "MPI: A message passing interface standard," vol. 8, 1994.

[5] Argonne National Laboratory, Mathematics and Computer Science, "The Message Passing Interface (MPI) standard," http://www-unix.mcs.anl.gov/mpi/.

[6] OpenMP Architecture Review Board, "The OpenMP specification for parallel programming," available at http://www.openmp.org.

[7] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of system overhead on parallel computers," in *The 4th IEEE Int. Symp. on Signal Processing and Information Technology (ISSPIT 2004)*, Rome, Italy, December 2004, http://bravo.ce.uniroma2.it/home/gioiosa/pub/isspit04.pdf.

[8] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Backman, "The ghost in the machine: Observing the effects of kernel operation on parallel application performance," in *SC '07: Proc. of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[9] 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 *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003.

[10] D. Tsafrir, Y. Etsion, D. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications," in *ICS '05: Proc. of the 19th Annual International Conference on Supercomputing*. New York, NY, USA: ACM Press, 2005, pp. 303–312.

[11] P. De, R. Kothari, and V. Mann, "Identifying sources of operating system jitter through fine-grained kernel instrumentation," in *Proc. of the 2007 IEEE Int. Conf. on Cluster Computing*, Austin, Texas, 2007.

[12] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA: ACM, 1967, pp. 483–485.

[13] P. De, V. Mann, and U. Mittaly, "Handling OS jitter on multicore multithreaded systems," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[14] K. B. Ferreira., P. G. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[15] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly, 2005.

[16] C. Boneti, R. Gioiosa, F. Cazorla, J. Corbalan, J. Labarta, and M. Valero, "Balancing HPC applications through smart allocation of resources in MT processors," in *IPDPS '08: 22nd IEEE Int. Parallel and Distributed Processing Symp.*, Miami, FL, 2008.

[17] C. Boneti, R. Gioiosa, F. Cazorla, and M. Valero, "A dynamic scheduler for balancing HPC applications," in *SC '08: Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[18] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "autopin: Automated optimization of thread-to-core pinning on multicore systems." in *Transactions on High-Performance Embedded Architectures and Compilers*, 2008.

[19] B. Rountree, D. K. Lowenthal., B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: Making DVS practical for complex HPC applications," in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 460–469.

[20] A. Duran, M. Gonzàlez, and J. Corbalán, "Automatic thread distribution for nested parallelism in openmp," in *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 121–130.

[21] E. Betti, M. Cesati, R. Gioiosa, and F. Piermaria, "A global operating system for HPC clusters," in *CLUSTER*, 2009, pp. 1–10.

[22] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *Proc. of the 2006 IEEE Int. Conf. on Cluster Computing*, Barcelona, Spain, 2006.

[23] T. Jones, S. Dawson, R. Neel, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10.

[24] P. Terry, A. Shan, and P. Huttunen, "Improving application performance on HPC systems with process synchronization," *Linux Journal*, November 2004, http://www.linuxjournal.com/article/7690.

[25] J. Liedtke, "Improving IPC by kernel design," in *SOSP '93: Proceedings of the 14th ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 1993, pp. 175–188.

[26] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995, pp. 251–266.

[27] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*. Orcas Island, Washington: IEEE Computer Society, May 1995, pp. 78–83.

[28] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole, Jr., "The operating system kernel as a secure programmable machine," *Operating Systems Review*, vol. 29, no. 1, pp. 78–82, January 1995.

[29] J. Appavoo, K. Hui, M. Stumm, R. W. Wisniewski, D. D. Silva, O. Krieger, and C. A. N. Soules, "An infrastructure for multiprocessor run-time adaptation," in *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM, 2002, pp. 3–8.

[30] 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," *Concurr. Comput.: Pract. Exper.*, vol. 21, no. 6, pp. 793–817, 2009.

[31] J. E. Moreira, M. Brutman, n. José Casta T. Engelsiepen, M. Giampapa, T. Gooding, R. Haskin, T. Inglett, D. Lieber, P. McCarthy, M. Mundy, J. Parker, and B. Wallenfelt, "Designing a highly-scalable operating system: the Blue Gene/L story," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 118.

[32] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 1–12, 2006.

[33] ——, "Operating system issues for petascale systems," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 2, pp. 29–33, 2006.

[34] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, no. 1, pp. 3–16, 2008.

[35] E. Shmueli, G. Almasi, J. Brunheroto, n. José Casta G. Dozsa, S. Kumar, and D. Lieber, "Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L," in *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 165–174.