A Global Operating System for HPC Clusters

Emiliano Betti*, Marco Cesati*, Roberto Gioiosa†, Francesco Piermaria*
*System Programming Research Group,
Univ. of Rome "Tor Vergata"
Via del Politecnico, 1 - 00133 Rome, Italy

†BlueGene Software Division,
IBM TJ Watson Research Center
1101, Kitchawan Rd - 10458 Yorktown Heights, NY, US

{betti,cesati}@sprg.uniroma2.it, rgioios@us.ibm.com, francesco.piermaria@gmail.com

***Abstract*—Modern supercomputers consist of clusters of thousands of independent nodes interconnected through fast networks. These nodes run independent operating system kernels, thus synchronization among them is demanded for user mode programs. This means that temporal synchronization of the nodes is a daunting task.**

On the other hand, HPC cluster applications often require a rather strict temporal synchronization for activities like performance analysis, application debugging, or data checkpointing. Therefore, the performance of an HPC parallel application may be severely impaired by the lack of temporal synchronization among the activities of the nodes of the cluster; this poses a severe limit on the scalability of such architectures. In this paper we introduce CAOS, an extension of the Linux kernel that aims to address the temporal synchronization problems of modern HPC clusters. We describe the general ideas behind CAOS, and we discuss some details of a possible implementation. We also illustrate some experiments performed on a prototype implementation of CAOS including a centralized network time tick, which allows a master node to synchronize the activities of all other nodes in the cluster, and a specific task scheduler tailored for HPC applications. These experiments, performed on a modern HPC cluster, witness that this new component has no measurable impact on the efficiency of the nodes while reducing the OS noise and providing better performance prediction. An implementation of CAOS based on this component can achieve a significant gain in terms of synchronization, global control, and scalability of the cluster.

I. Introduction

The area of computer science aimed at designing, handling, and programming cluster-based supercomputers is nowadays referred to as *High Performance Computing* (in short, HPC). Typically, a modern HPC supercomputer is composed of a collection of thousands of nodes equipped with *Commercial, Off-The-Shelf* (COTS) processors and interconnected by specialized, fast networks.

In this paper we introduce *CAOS*, *Cluster Advanced Operating System*, a prototype for an HPC Operating System (OS) based on Linux. The main idea behind CAOS is quite simple: to synchronize the activities of all cluster nodes. Rather than having several independent kernel programs controlling the nodes of the cluster, CAOS synchronizes the activities of the nodes by removing the local metronomes (local timers) and replacing them with a unique global time source for all the nodes generated by a master node. In this way, all the activities performed on each node can be synchronized with a global metronome. As a result, many scale activities can be executed simultaneously.

From the software architecture point of view, CAOS is essentially a global operating system that consists of two components: a *global orchestra conductor* running on a *master node* and *local instrumentalists* running on the *computing nodes*. The global conductor coordinates all computing nodes and notifies each of them with the activities that should be performed. The local instrumentalist actually executes the operations when notified by the conductor.

Upon this global timing mechanism CAOS implements a protocol aimed at scheduling global and local activities simultaneously. Thus, CAOS provides the low-level features that allow the developer to create more powerful tools. As an example, a debugging tool for parallel applications can take advantage of the capability to stop the whole parallel application at once; nowadays, this simple operation cannot be done reliably.

We performed experiments on a single node to show how CAOS reduces the OS noise caused by the timer interrupt and the scheduling of other, low-priority processes. Moreover, since the synchronization process may delay some tasks or system activities thus potentially causing performance degradation, we used classic NAS benchmarks, which are widely adopted in the HPC community, to perform some preliminary tests on a prototype implementation of the conductor-instrumentalist mechanism that replaces the local metronomes. The results show that there is no measurable loss in performance. On the other hand, CAOS may be a significant gain in terms of synchronization, global control, and scalability.

The rest of the paper is organized as follows: Section II gives some insight on modern HPC clusters, describes the characteristics of HPC applications, and explains the main reasons behind the design of CAOS. Section III describes the general design of CAOS. Section IV briefly describes HPCSCHED, which is a task scheduler aimed at HPC applications. Section V provides information about the implementation of a prototype heartbeat mechanism for CAOS (NETTICK). Section VI shows our preliminary experiments. The related work is described in Section VII. Finally Section VIII describes our conclusions and suggests future works.

978-1-4244-5012-1/09/$25.00 ©2009 IEEE

II. HPC SYSTEMS AND APPLICATIONS

Many scientific, commercial, and military activities require very high computational power; i.e., consider theoretical physics, weather forecasting, or cryptanalysis, to name just a few applications. Historically, supercomputers were dedicated machines aimed at solving specific kinds of computationally challenging problems. These machines were really expensive and only a few institutions were able to own one of them.

Progress in the computer industry and in computer science has radically changed this scenario. Nowadays, it is possible to build a supercomputer by means of high-end *Commercial, Off-The-Shelf* (COTS) computers grouped in *clusters*. These modern supercomputers can be assembled at a fraction of the cost of the ancient supercomputers and have generally much more computational power.[1] As a matter of fact, almost all supercomputers built in the last ten years consist of HPC clusters. The last (November 2008) TOP500 Supercomputer Sites list [22] includes 410 HPC clusters out of 500 total entries. Each node of an HPC cluster is generally a *Symmetric Multi-Processor* (SMP) system with one or more processor chips. Moreover, each processor chip often includes a multi-core and/or multi-thread processor. Summing up, a supercomputer may easily have tens of thousands of computational cores contained in a large number of computing nodes interconnected by a high-speed network.

Controlling such a number of computational units is a daunting task. The OS commonly used in these supercomputers runs a version of the Linux kernel (439 entries out of 500 in the Top500 list). Typically, each node of the cluster executes a Linux kernel that handles all cores in the node. However, the kernels running on the nodes are very loosely coupled.

Most of the HPC applications running on supercomputers are *Single Process-Multiple Data* (SPMD) and are usually implemented using an MPI library [1] or an OpenMP library [18] (or a combination of both of them). Nowadays, MPI applications are more common, since shared memory machines with a lot of processors are still very expensive. This situation might change in a near future, for chips with tens or hundreds of cores will likely soon appear on the market. Nevertheless, as most of today's applications are based on MPI, we choose to focus on the typical behavior of MPI parallel programs.

MPI applications can be characterized by a cyclic alternation of two phases: a *computing phase* in which each process performs some computation on its local portion of the data set, and a *synchronization phase* where processes communicate by exchanging data among themselves. These two phases usually interleave, and this behavior repeats till the end of the application. Crucial for performances is ensuring that all nodes start and terminate each phase at the same moment, so as to minimize the time wasted by the faster nodes while waiting for the slower nodes to terminate the current phase.

Achieving this goal is not easy, though. An important source of degradation in big MPI applications is the so-called *operating system noise*. Essentially, the activities of the operating system kernels—mainly, the timer interrupts in number crunching applications—induce some delays on the MPI processes during the computing phases. The OS noise, as well as its effects on parallel applications, has already been analyzed and measured, see for example [8], [11], [17], [19], [23]. It is important to observe, however, that while the noise in a single node only marginally affects the performance (about 1–2% on average), its effects in a large scale cluster may become dramatic. In fact, when scaling up to thousands of nodes, the probability that in each computing phase at least one node is being slowed down by a long kernel activity approaches the value 1—thus drastically reducing the parallel application performance. This phenomenon is called *noise resonance* [11]. Figure 1(a) shows how the OS noise affects the performance.

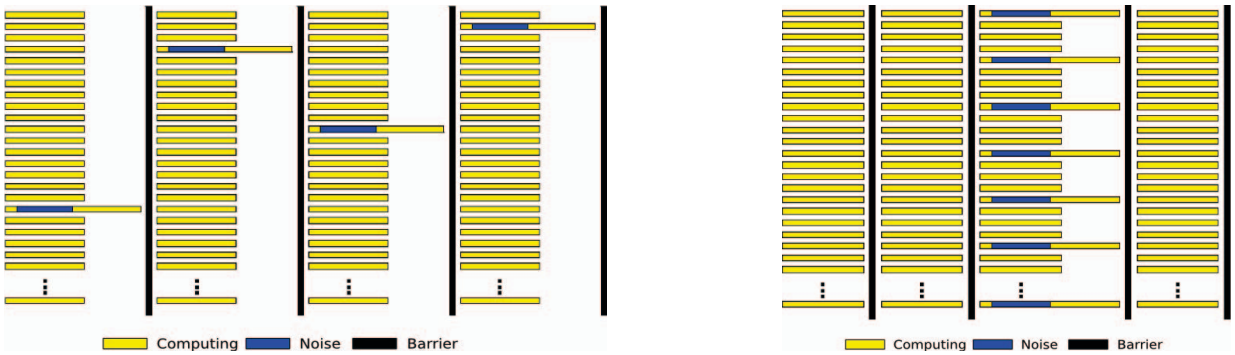
As a practical example, consider that by default the Linux kernel writes dirty pages on disk every five seconds: these timeouts usually expire at different moments on each node, mainly because of the small differences in the frequencies of the internal clocks. Although these operations are necessary for the correct functioning of the machine, a cluster-aware OS might minimize the impact on the OS noise by forcing the cluster nodes to perform all disk cache flush operations at the same time. Therefore, the slowdown affects just one computing phase every five seconds rather than, statistically, *all* computing phases in five seconds. Figure 1(b) shows how coordinating the OS activities increases the performances.

Global time synchronization has always been a daunting task for supercomputers. COTS-based supercomputer still have to face the problem that the cores of each node of the cluster have their own timers, not synchronized with all the others. In this scenario is clear that, since the "time" is not the same on different nodes, OS activities and synchronization is difficult to achieve. Some of the custom supercomputers try to solve this problem by means of high precision timer metronomes. BlueGene/L (and later its successor, BlueGene/P) is the first example in this direction: the timers of each node of the supercomputer are continuously synchronized and, as a result, the "time" is the same in each node, i.e., each processor's time base register contains the same value. This mechanism greatly simply the "cluster-based" activities: for example, it is possible to profile all the processes running on each node and then merge the traces because there is no skew between the time registers. From this point of view, it is like if the supercomputer were composed of one big node. Moreover, the timer system, together with the *Computer Node Kernel* (CNK) micro-kernel, reduces the OS noise virtually to zero.

The solution adopted for BlueGene supercomputers is custom for these machines and it is not available for COTS clusters. NETTICK described in Section V-B, aims at providing a less expensive, similar mechanism, though not as precise, for a general cluster.

Besides improved performance, users of HPC clusters may expect to apply to the whole cluster some services provided by the local operating systems, such as activities related to

[1] There are a few demanding computational problems that cannot be efficiently solved by splitting the computation among the nodes of a cluster. For this kind of application, dedicated supercomputers are still built.



(a) Possible effect of OS noise

(b) CAOS coordination of OS noise

Fig. 1. The impact of OS noise on parallel applications

performance analysis or debugging. Thus, some local services should be extended to scale up with the number of nodes in the cluster. For example, as an operating system kernel for a single node provides the mechanism to stop the application and have a look at the executed code or the referenced data set, an operating system for HPC should provide the capability to stop the parallel application (at the same moment on all nodes) and let the user take a coherent snapshot of what is going on. Fault tolerance is another important service that requires a strict synchronization to provide correct results. For example, the *Mean Time Between Failure* (MTBF) measure of a single node is large enough that data checkpointing is usually not required. However, a fault tolerance mechanism is mandatory in large clusters because their global MTBF is much smaller—this is obvious when considering the number of components that composes such a cluster.

To sum up, the scenario is the following: though neither the hardware nor the software were originally intended for High Performance Computing, HPC clusters are widely used nowadays and show the fastest growth among the architectures for supercomputers. These supercomputers consist of tens of thousands of (virtual) CPUs, each one potentially running an HPC process on top of a sophisticated hardware and of a local operating system kernel. It should not be really surprising that managing, coordinating, and synchronizing this enormous number of processors and processes poses real-life problems that are still largely unsolved.

III. CAOS

CAOS (*Cluster Advanced Operating System*) is a global operating system for HPC clusters based on Linux. Through this paper we will refer to version 2.6.24 of the Linux kernel. The core layer of CAOS essentially consists of two main agents:

Global conductor: This agent, which runs on a selected master node, is in charge of scheduling cluster activities. One of its main jobs is to provide a time source (*heartbeat*) to the cluster nodes. Furthermore, the conductor notifies the cluster nodes about global and local system activities. Local system activities are related to the proper functioning of the cluster node. Thus, the conductor may command the nodes to check for decayed software timers, reclaim pages from the disk caches, flush the disk caches, perform a forced process switch, and so on. Global system activities are related to the extended functionalities provided or supported by CAOS, such as tracing and debugging the parallel applications, data checkpointing, and performance analysis. The global conductor can be implemented as a user mode program running on the master node. In fact, any available job schedulers could be easily modified in order to provide the required synchronization protocol.

Local instrumentalist: This agent runs on each cluster node. Its main job is to perform the operations scheduled by the global conductor and, in some cases, to notify the global conductor when operations have been successfully completed. The main design goal of the local instrumentalist is to not interfere with the computing phases of the HPC applications, unless instructed by the conductor to do so. The main component is the kernel's task scheduler. CAOS makes use of an improved version of the Linux task scheduler named HPCSCHED [5], which has been specifically designed for improving the performance of HPC applications by addressing some of the typical HPC problems (i.e., load imbalance).

The global conductor provides a heartbeat event every N seconds, where N can be either determined by the HPC application programmer or dynamically adjusted by CAOS itself. The heartbeat consists of broadcast frames sent on a dedicated network channel that connects all cluster nodes. The payload of these frames may also contain the conductor's commands that force the nodes to schedule the system activities.

The master node may easily become a bottleneck for the performance of the clusters. We thus designed this protocol to be unidirectional: the global conductor schedules the activities to be executed and notifies the cluster nodes; however, it does not wait for their completions. This design choice improves scalability, because the network underneath the nodes is usually capable of sending a broadcast message within a fixed amount of time regardless of the number of nodes that will receive the message. This is true for many high-end networks adopted in HPC clusters, e.g., Myrinet, Quadrics, or Infiniband. Unfortunately, a pure unidirectional protocol is sometimes not sufficient, because some large scale activities are considered completed correctly only when all nodes in the

cluster have successfully terminated the operation. For example, a data checkpoint is valid only if all cluster nodes have successfully completed their checkpoint, otherwise it must be discarded. In these cases, a more sophisticated bidirectional protocol is required, which allows the master to be notified about completions of cluster node tasks.

In the next two sections we will describe HPCSCHED—the task scheduler—and NETTICK—our prototype implementation of the heartbeat mechanism.

IV. THE HPC SCHEDULER

HPCSCHED is a new scheduler designed for HPC clusters. It is capable of balancing MPI applications using the hardware resource allocation mechanism provided by an underneath processor (like the IBM POWER6), and of minimizing OS noise caused by other user and kernel threads. HPCSCHED is extensively described in [5]; here we briefly highlight some of its features.

HPCSCHED is implemented inside the Linux kernel as a new scheduler for HPC applications. Since we want to prioritize HPC over normal processes, we placed the HPCSCHED scheduling class between the two standard Linux classes Real-Time and CFS

In order to balance the HPC application, the scheduler tracks the application behavior and detects when to increase or decrease the amount of processor's internal resources assigned to a specific process. The scheduler is also easily extensible and can be integrated with other solutions useful for HPC applications, such as a checkpoint/restart mechanism [12].

The HPC scheduler is based on three components, mainly independent from each other:

Scheduling policy: The scheduler algorithm used by the Scheduler Core to select the next task to run among the runnable tasks in the HPC class.

Load Imbalance Detector and Heuristics: We use a Load Imbalance Detector and heuristic functions to select, according to the scheduler metrics, the amount of hardware resources to assign to the task.

Mechanism: Architecture-dependent, utility functions necessary to manage the amount of hardware resources assigned to a task.

A. The Linux Scheduler Framework

The Linux kernel 2.6.23 introduced a process task scheduler named *Complete Fair Scheduler*, CFS), and a *scheduler framework*, which divides the scheduler in two main components: three *Scheduling Classes*, which implement the policy details, and a *Scheduler Core*. The Scheduling Classes are *objects* that provide suitable methods for any low-level operation invoked by the Scheduler Core (for example, selecting the next task to run or accounting for the time elapsed). Each of the three Scheduling Classes contains one or more scheduling policies. In order to improve scalability, each CPU has a list of Scheduling Classes. Each class, in turn, contains a list of runnable processes belonging to one of the policies handled by the class. The first class (the highest priority) contains real-time processes (`SCHED_FIFO` and `SCHED_RR`); the second class (the new CFS class) contains the normal processes (`SCHED_NORMAL` and `SCHED_BATCH`); finally, the last class contains the idle process (`SCHED_IDLE`). The ordering of the Scheduling Classes introduces an implicit level of prioritization: no processes from a low priority class will be selected as long as there are available processes in one of the higher priority classes. It follows that the idle process will never be selected if there are runnable processes in other classes.

B. HPCSCHED *design*

Taking advantage of the new scheduler framework described in the last section, HPCSCHED introduces a new Scheduler Class (`sched_hpc`) and a new scheduler policy for HPC applications (`SCHED_HPC`). A user can move an application to the HPC class by means of the standard `sched_setscheduler()` system call.

The typical way of running MPI applications on current supercomputers is to run one MPI process per-core or hardware thread (corresponding to a logical CPU in Linux). Thus, we expect to have one process in the HPC class of every CPU (maybe two or three during workload balancing). Under this assumption, it is not worthwhile to have a complex algorithm for selecting the next task to run. In fact, given this small number of processes in the run-queue list, a simple round-robin list is as good as a more complex red-black tree. However, the code for a round-robin run-queue is much simpler and with better performances (for example, the scheduler does not have to balance any tree). The scheduling policy is, however, independent of the other components and it can be changed without affecting the other components.

In the new Linux kernel framework, workload balancing, i.e., evenly splitting the workload among all the available processor domains [6] at core-, chip- and system-level, is also performed inside the Scheduling Class level. Every Scheduling Class has its workload balancing algorithm, which means that each CPU has roughly the same number of real-time or normal tasks.

The workload balancer is invoked whenever the kernel detects that there is a big imbalance or if one processor is idle. In the latter case, the idle CPU tries to pull tasks from other, busier run-queue lists to its run-queue.

The HPC workload balancing algorithm makes sure that each processor domain runs the same number of processes. For example, in a dual-core, 2-way SMT system there are three domain levels: chip level, core level and hardware thread level (a hardware thread is what is recognized by the OS as a CPU). Our workload balancer tries to balance the number of task at each domain level. Thus, a core domain running fewer tasks than another core will try to pull tasks from the other core.

As already mentioned, MPI applications alternate *computing phases* (when a process is runnable) with *synchronization phases* (when a process is not runnable because of waiting for an incoming message or for synchronization). HPCSCHED considers the sum of a computing phase and of a synchronization phase as one *iteration* of the MPI application.

The scheduler learns from the execution history of a process: the general idea is that if a task does not have a high CPU utilization during iteration i, it will perform in the same way in the $i+1$ iteration. This is a common case, for example, for those applications that compute an approximation of a solution of a problem and then try to reduce the approximation error. The Load Imbalance Detector assumes that iteration i is representative of iteration $i+1$, hence, at the end of iteration i, the HPC scheduler tries to balance the application changing the amount of hardware resources assigned to a task and assign a different amount of resources before iteration $i+1$ starts. In the POWER5 and POWER6 implementation of our scheduler, HPCSCHED uses the *hardware thread priority mechanism* to assign more or less hardware resources to the processes. This is the only part of the scheduler that depends on the processor architecture. The main idea is that the higher the hardware thread priority, the higher the amount of hardware resources assigned to the process, the higher the performance of that process. The particular mechanism works at fetch level: the core fetches more often from the higher priority thread.

How good is our solution strongly depends on how this guessing is close to the optimum solution. If the guessing is not correct, in iteration $i+1$ the application may become even more imbalanced than in iteration i. Hopefully, the scheduler will detect this anomaly during iteration $i+1$ and apply the right resource balancing in iteration $i+2$. Clearly, not all applications present a well defined iterative structure with a barrier at the end of the iterations, in which case HPCSCHEDmay fail to balance the application, and new heuristics are likely to be required.

The scheduler may require some iterations to converge to a balanced solution: the goal of the heuristic is to find a stable state where the application is balanced and to remain there as long as the application behavior is constant.

Fig. 2. HPC application iterative behavior

While a task is running, the scheduler collects several metrics, such as the task's execution and waiting time. Figure 2 shows a typical task trace: the process computes for t_R seconds and then goes to sleep, waiting for messages coming from the other processes in the MPI application (t_W). If $t_i = t_R + t_W$ is the total execution time in iteration i, then the task utilization in the same iteration is $U_i = t_R/t_i$. The global task utilization is the ratio of the accumulated running and iteration times: $U = \sum t_R / \sum t_i$. These metrics are quite easy to compute, since the kernel already provides some of the required values. We only had to add the values necessary to introduce the concept of *iteration* that is not present in the standard Linux kernel.

Once the information about the task's progress has been stored, the HPC scheduler has to heuristically decide whether to increase, decrease or keep the same amount of hardware resources assigned to the current process in the next iteration.

V. NETTICK

A. Linux timekeeping architecture

Version 2.6.24 of the Linux kernel defines two types of time devices: clock source devices and clock event devices. The *clock source device* represents a hardware mechanism that keeps track of time. Typical examples are the Time Stamp Counter (TSC) of the Intel and AMD architectures, and the High Precision Event Timer (HPET) chip. The *clock event device* represents a hardware mechanism that can be programmed to raise an interrupt after a specified amount of time. Typical examples are the APIC Local Timer of Intel and AMD processors, or the Decrementer of PowerPC processors. It should be noted that some hardware devices can be used both as clock source and as clock event devices.

Modern computers include many different time-related devices. Each of them has its peculiar characteristics, like resolution and accuracy, reprogramming costs, and so on. Moreover, each device has a statically assigned *rating* that represents a measure of its reliability and convenience. The Linux kernel selects the top-rated clock source device and clock event device, which will be used for all time-related activities. This decision, however, is not set in stone: the rating of a clock source device can be lowered if the kernel determines that the circuit is not very reliable after all. For example, this happens quite often with unstable Time Stamp Counters in chips with voltage and/or frequency scaling. Thus, the kernel may decide to elect another top-rated clock source device. (Top-rated clock event devices are never replaced unless a new, better hardware device is registered.) A reference to the top-rated clock event device for a given CPU is contained in a so-called *tick device*, which is a per-CPU object whose methods can be used to program a timer event.

The Linux timekeeping architecture is based on *tick events*. Basically, a tick coincides with the occurrence of a periodic timer interrupt raised by a clock event device. The number of ticks elapsed since the bootstrap is stored in `jiffies_64` global variable. At each tick, the kernel executes some specific operations. If the CPU has a Local APIC unit that can raise local timer interrupts (this is true in almost all multiprocessor systems), the timer interrupt handler executes some specific per-CPU operations like, for example, checking for expired software timers. There might also be a global clock event device that raises global timer interrupts.

The timekeeping architecture provides two operating modes: the *periodic mode* and the *dynamic tick mode*. Moreover, the kernel may be compiled to support the high resolution timers in both operating modes. In *periodic mode*, the kernel programs the tick device once and for all in such a way to raise a periodic interrupt with a frequency defined at kernel compile time (macro `HZ`). In this case, `jiffies_64` will be incremented by one at every tick event. In *dynamic tick* mode the kernel programs the tick device in "one shot mode", that is, in such a way to raise a single timer interrupt when the first event of interest occurs. The next event may be an infra-tick event (e.g., the activation of a high resolution timer,

if supported) or the next tick event, whichever comes first. Furthermore, in dynamic tick mode a tick event is programmed only if the CPU is busy. If the CPU is idle, the tick device is programmed so as to raise an interrupt at the occurrence of the first decaying software timer, if defined.[2] Therefore, the time interval between two tick events may be more than the canonical tick period ($1/\texttt{HZ}$), and the `jiffies_64` variable has to be adjusted by a factor depending on the time returned by the top-rated clock source device and according to the formula $\texttt{HZ} \times (\text{now} - \text{last } \texttt{jiffies\_64} \text{ update})$.

B. Implementation of NETTICK

NETTICK is our prototype implementation of the heartbeat mechanism of CAOS. It is implemented as a patch for the Linux kernel version 2.6.24.

NETTICK basically defines a new per-CPU virtual clock event device, the so-called *network event device*. While real clock event devices raise a timer interrupt when the hardware circuits determine that the proper amount of time since the last programmed event has elapsed, the network event device raises a timer interrupt when a special frame arrives on a specific network device.

The current implementation of NETTICK relies on heartbeat signals embedded on a special type of IEEE 802.3z (Gigabit Ethernet) frames. The choice of the Gigabit Ethernet was motivated by several facts. Firstly, we ran our experiments on an HPC cluster based on Gigabit Ethernet network devices only (see Section VI). Secondly, according to the TOP500 list [22], more than 56% of the top-ranking HPC clusters are based on a Gigabit Ethernet. Thirdly, if NETTICK can be shown to perform well on a Gigabit Ethernet, it can presumably work with any other network architecture having similar or higher performance.

Specifically the Linux kernel has been modified as follows:

- When NETTICK is initialized, new clock event devices are registered, one for each available CPU. Initially the network clock devices have a default rating equal to zero; thus the kernel does not change the current top-rated clock event device (that is, the tick device).
- Through the */sysfs* filesystem, the administrator can set a new value for the rating of the network event devices. Any change forces the kernel to evaluate the hierarchy of clock event devices and, if appropriate, to change the current tick device.
- A new type of Ethernet frame (`0x88CB`, denoting the heartbeat special frame) is registered, together with a corresponding `nettick_rcv()` handler function. In order to use a different family of network devices, it is sufficient to register another type of frame; the frame handler does not have to be changed. The Linux networking stack ensures that the handler function will be invoked once per each frame arriving from the network. Thus, at each heartbeat `nettick_rcv()` is invoked (in software interrupt context).
- The `nettick_rcv()` function just sends an Interprocessor Interrupt (IPI) to all available CPUs of the node.[3] Notice that this step is required to perform kernel activities on the CPUs that did not received the interrupt from the network card.
- The interrupt handler associated with the NETTICK IPI simply invokes the event handler of the tick device.

When a network event device becomes the new per-CPU tick device, the old tick device (e.g., the Local APIC for Intel/AMD and the Decrementer for PowerPC) is switched off.

The kernel then associates the function to handle a tick with the network event device. For instance, if the kernel does not support high resolution timers and it runs in dynamic tick mode, then it associates the `tick_nohz_handler()` function with the current tick event handler. Receiving a heartbeat frame triggers the activation of `nettick_rcv()`, which in turn sends in broadcasting an IPI to all CPUs; finally, the IPI handler executes the event device handler of the tick device, thus triggering the execution of the usual tick-related kernel activities.

It should be noted that the NETTICK module works properly only if the kernel runs in dynamic tick mode. The reason is that CAOS must be able to issue heartbeat frames at custom or variable rate, while in periodic mode the timekeeping architecture assumes that the ticks have frequency exactly `HZ`. Therefore, at every timer interrupt `jiffies_64` is incremented at most by one (or by two, on very rare occasions). As we have seen, in dynamic tick mode the kernel increments `jiffies_64` according to the time effectively elapsed since the past update of the variable.

VI. EXPERIMENTS

In this section we present our experimental results based on the current prototype version of CAOS. We ran three sets of different experiments, two of them (Section VI-A and Section VI-B) show how CAOS reduces the OS noise inside a single node using both NETTICK (timer interrupt noise) and HPCSCHED (scheduling noise). In Section VI-C we show that deferring the OS activities related to the timer interrupt does not impact on the performance of parallel applications.

A. NETTICK

In order to analyze the deterministic processing behavior of the OS and, therefore, the noise affecting a node, we ran *Fixed Time Quanta* [10] (FTQ). FTQ measures the amount of work done in a fixed time quantum in terms of *basic operations*. The time quantum is quite short (around one microsecond). Periodically, at each time interval T, the benchmark samples how many basic operations were performed in the last sample and reports them. Clearly, the difference between the maximum number of basic operations, N_{max} (about 31300 in our

[2] Actually things are more complicated, because the registers of the clock devices have limited capacity. Thus there is a maximum time interval delay that can be effectively programmed.

[3] The association of an IPI number with the handler is the only architecture-dependent part of NETTICK. We have successfully implemented it for the IA32, AMD64/EM64T, and PowerPC 64 architectures.

case), and the number of basic operations in a sample i, N_i, is due to other activities external to the application performed by the OS, i.e., noise.

The FTQ benchmark was executed on a single quad core 2.66GHz Intel(R) Xeon(R) 5150 node, with 4-GB of RAM and two IEEE 802.3z (Gigabit Ethernet). In this first experiment we analyzed the OS noise introduced by the timer interrupt. As reported in [8], [11], [23], the timer interrupt is the main source of OS noise (in [8] the authors claim that the timer interrupt accounts for 63% of the OS noise). In this experiment we ran our test in a similar condition as in [11]: the machine was idle (no other process running during the test session) and we reduced the number of kernel threads and user daemons running. Even if the experiment is not representative of a real environment (user and kernel daemons may be running), it is useful to isolate the effect of the timer interrupt. In the next sections we will show the results of the experiments performed on a real cluster of HPC nodes.

Figures 3(a) and 3(b) report the number of *missing basic operations*, i.e., the difference between N_{max} and N_i. Another way to look at these results is to consider each bar as the duration of the OS noise measured in terms of basic operations.

In Figure 3(a) we can observe different kinds of noise: the very small and very frequent noise (20-50 basic operations) is the timer interrupt. What happens here is that the local timer raises an interrupt and the kernel executes the associated interrupt handler only to discover that there are no pending operations to be performed. This is a case we can safely eliminate using NETTICK: Figure 3(b) shows how this noise has been consistently reduced (we will see in Section VI-C that this approach does not impact on the performance of a parallel application). The second kind of noise in Figure 3(a) (650 basic operations), as well as the third (more than 6500 basic operations) are related to network tasklets. In [11] the authors also identify the network tasklets as a second source of OS noise, less frequent but larger than the timer interrupt noise. We repeated the same experiments disconnecting the machine from the network and observed as those spikes disappeared. For fairness with the experiments performed in Section VI-C and for making the experiments more representative (the node is part of a cluster connected through a network), we omit these results.

As Figure 3(b) shows, the last kind of OS noise is not eliminated by NETTICK: those kernel activities are, in fact, necessary for the correct functioning of the machine, hence, cannot be simply removed. In order to reduce the OS noise at scale, those activities have to be co-ordinated by the Global Orchestrater.

B. HPCSCHED

The results in this section are based on the same kind of experiments of the previous test; however, here we also introduce user daemons that perform cluster activities (statistics, parallel file system, checkpoint/restart, etc.). User daemons are quite common in HPC clusters and may introduce another kind of noise, less frequent but orders of magnitude higher than the timer interrupt. As reported in [4], in fact, the OS may decide to schedule one of those user daemon instead of the HPC application, thus delaying the computing phase of one or more processes and creating load imbalance.

In this experiment we introduce a user daemon that collects statistics and sends them to the job manager. In order to strictly evaluate the impact of the noise introduced by the scheduler, we assume 0-overhead for collecting statistics, i.e., the user daemon does not collect any statistic but periodically wakes up and send a message to the job manager. In this way the noise of the statistic collector is limited to the OS scheduler; this is an ideal case, because in a real HPC system there would be many more user daemons and the noise introduced by each of them would be higher.

However, even in this simplistic test case, the amount of noise increased considerably, as Figure 4(a) shows. With respect to Figure 3(a), in Figure 4(a) there is another kind of periodic noise (around 3500 basic operations) due to the fact that the OS was running the statistic collector instead of the FTQ application. As we can see in Figure 4(b), HPCSCHED reduces this noise preventing the statistic collector from interrupting the parallel application during the computing phase. The scheduler will run the statistic collector during the synchronization phase, i.e., when the HPC application is communicating and does not need to use the CPU.

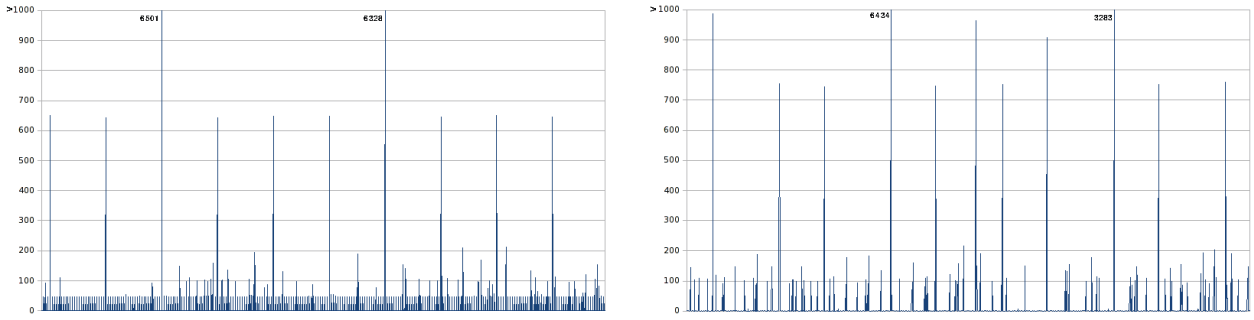
C. Scaling on a Cluster

By removing the timer interrupt some operations might be delayed. In this section we show that, even though we reduced the OS noise caused by the timer interrupt, the performance of HPC applications was not affected. We performed some experiments to evaluate the performance of a cluster that uses the NETTICK global heartbeat. For this test we used a cluster of 24 Apple Xserve. Each node has 2 microprocessors Intel(R) Xeon(R) 5150 with a frequency of 2.66 GHz. These chips are dual core, so every node has 4 cores and the whole cluster has 96 cores. Each node includes 4 GB of RAM and is connected to the other nodes by means of a IEEE 802.3z (Gigabit Ethernet) network. On each node is installed a 64-bit Linux distribution and Open MPI 1.2.5.

Using a Gigabit Ethernet yields perhaps the worst possible scenario for our test, essentially because large Ethernet networks must use a hierarchy of bridge devices (switches) that introduce significant delays.[4] Moreover, heartbeat frames can be queued inside the kernel or the network card device, because they compete with the other traffic on the network. Thus, we think that in a real application scenario NETTICK must be granted exclusive access to a dedicated network channel. Typical HPC nodes include two network cards, so this can be easily arranged. We stress again that NETTICK can be adapted so as to use any kind of high-speed network, and many network families like Infiniband, Quadrics, Myrinet or BlueGene Torus do not have the drawbacks of Gigabit Ethernet.

We performed the tests by running 4 times some of the benchmarks included in the widely used MPI-based NAS

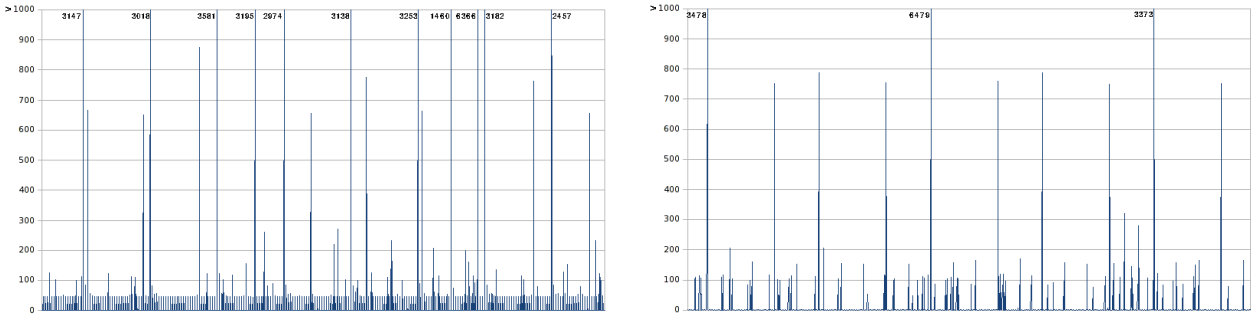
[4]Gigabit Ethernet networks typically use full-duplex, point-to-point links, so latencies due to the medium access control protocol are not really an issue.



(a) FTQ with standard system

(b) FTQ with NETTICK

Fig. 3. Timer Interrupt noise in FTQ



(a) FTQ with standard system

(b) FTQ with NETTICK

Fig. 4. Scheduler noise in FTQ

Parallel Benchmark (NPB) suite, version 2.4 [16]. We selected two benchmarks: *EP* and *BT*.

EP Embarrassing Parallel benchmark. This benchmark is supposed to scale linearly with the number of nodes in the cluster. In fact, the application consists of processes that do not require communication to complete their job, hence adding a new node linearly reduces the amount of per-processor work, thus reducing the total execution time. This benchmark has been executed on 4, 8, 16, 32, and 64 cores of our test cluster in order to analyze the scalability of the heartbeat mechanism.

BT The Block Tridiagonal Benchmark solves multiple independent systems of non-diagonally dominant, block tridiagonal equations with a 5x5 block size. This benchmark is representative of computations associated with the implicit operators of CFD codes (such as ARC3D at NASA Ames). Because this benchmark must run on a number of cores that is an exact square, we executed it on 81 cores of our test cluster. Further information about the test programs can be found in [2].

Clearly the performance of each application depends on the frequency of the heartbeat provided by the master. We therefore tested the following four configurations:

sys:	Vanilla kernel with system tick at 100 Hz
net10:	NETTICK kernel with heartbeat tick at 10 Hz
net50:	NETTICK kernel with heartbeat tick at 50 Hz
net100:	NETTICK kernel with heartbeat tick at 100 Hz

One node is used as master node, so it sends the heartbeat frames (in broadcast) and starts the MPI benchmark.

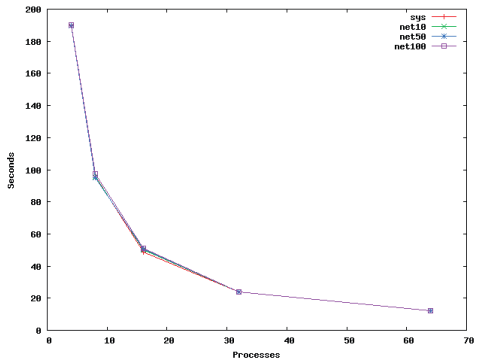


Fig. 5. Curves of execution times of NAS EP varying the number of cores

Type of test	Cores	sys	net10	net50	net100
EP	64	12.12	12.05	12.10	12.02
BT	81	194.78	194.39	194.52	194.76

TABLE I
TOTAL TIME IN SECONDS, MEASURED BY NAS PARALLEL BENCHMARK

We expected the *net100* configuration to be a bit slower than the standard *sys* configuration. In fact, in the former case, the number of interrupts per second was essentially the same as in the latter case. However, the overhead introduced by the network interrupt handler is larger than the overhead introduced by the local timer interrupt, hence the OS noise could delay the application more than usual. When running the test, we discovered that it was not possible to notice

this extra overhead: in fact, in some cases the heartbeat overhead was paid by all the nodes at the same time because of our global synchronization. This is not the case for the standard configuration, where the overhead introduced by each uncoordinated local timer interrupt has to be summed up (in the worst case).

Cores	sys	net10	net50	net100
4	190.24	189.87	189.81	189.95
8	96.18	95.46	94.90	97.44
16	48.70	50.20	50.40	50.85
32	24.06	24.00	24.01	23.88
64	12.12	12.05	12.10	12.02

TABLE II
EXECUTION TIME OF NAS EP (CLASS C) VARYING THE NUMBER OF CORES (TOTAL TIME IN SECONDS)

Table I lists the average of the four runs of the total time in seconds measured by the NAS parallel benchmarks EP and BT (class C). As we can see from Table I, there is no loss in using our proposal. All the results can be considered approximately constant, and the small differences are due to the coarse-grained time measurement of the benchmark.
Figure 5 shows how the performance of EP varies with the number of processes in the application. All curves in Figure 5 overlap, thus we report also the execution time of the experiments in Table II.
The test results confirm that the real speedup does not depend on the number of nodes. This means that the heartbeat mechanism has no negative effect on the performance or the scalability of the parallel application.

VII. RELATED WORK

The problem of achieving a tight synchronization of the activities of a number of components connected by some communication links has been extensively studied in the past, mostly for measurement and control systems.
According to [9] there are three possible strategies for achieving a tight time synchronization across separate components:

1) Message-based systems: timing is provided by special messages sent over the communication link.
2) Cycle-based systems: timing is based on a periodic scheduling natively provided either by the communication link or by a controlling device.
3) Time-based systems: timing is provided by real-time clocks inside all components. In distributed environments the local clocks must be synchronized by means of a suitable protocol such as the IEEE 1588 standard.

The IEEE 1588 standard, usually known as *Precision Time Protocol*, was originally published in 2002 [13] and is nowadays widely adopted. If implemented on top of suitable hardware communication devices, it can achieve a sub-microsecond precision. The protocol can be easily implemented by the nodes of an HPC cluster. For instance, Linux-based systems may use the open-source PTPd user daemon [7], [20]. This daemon receives time-stamped UDP or IP messages from a master node and computes a precise estimate of the offset between the master's clock and the local clock. The user daemon can then adjust the local clock of the node by means of the `adjtimex()` system call. Another open-source project aimed at synchronizing the local clocks of the nodes in an HPC cluster is BTime [14], [15], distributed by the Los Alamos National Laboratory. The design of BTime is very similar to the design of PTPd: the daemon exchanges UDP packets with a master reference node, computes the time offset with the master clock, and adjusts the local clock via the `adjtimex()` system call.
It should be noted that NETTICK is quite different from both PTPd and BTime. The main differences are: 1) In the above classification, NETTICK can be considered a message-based system, while PTPd and BTime are time-based systems. This essentially means that NETTICK achieves time synchronization by means of the messages themselves rather than by the value of local clocks adjusted by a user daemon. 2) NETTICK makes use of level 2 broadcast frames (Gigabit Ethernet in our prototype implementation) rather than UDP packets. We argue that in NETTICK there is no real advantage in using tick messages of level 3 (e.g., IP) or 4 (e.g., UDP) because PTPd and BTime use broadcast IP addresses that are embedded in broadcast level 2 frames. Furthermore, while IP packets can be routed across several level 2 networks, in practice the level 3 routers would introduce excessive delays. 3) Neither PTPd nor BTime offer any mechanism to synchronize the activities of the OS running on the nodes of the cluster. Besides the format of the packets, the actual constraint is that both protocols are implemented by user-level daemons. NETTICK is instead implemented at kernel level, so it is easy to extend it and embed in the heartbeat frames coming from the global conductor the commands that will be directly executed by the kernels of the nodes.
Operating system noise has been extensively studied in the past. In the HPC community this problem was first shown by Petrini et al. [19]: they explained how the OS noise and other system activities (not necessarily at OS level) dramatically impact on the performance of a large cluster. In [19] the authors observed that the impact of the system noise when scaling on 8,192 processors was so large (because of the so-called "noise resonance") that leaving one processor idle to take care of the system activities lead to a performance improvement of 1.87x. The impact of the operating system on classical MPI operations, such as collective, was examined in [3]. Though in [19] the authors did not identify every single source of OS noise, a following paper [11] identified timer interrupts (and the activities started by the paired interrupt handler) as the main source of OS noise. Other papers [8], [23] found the same result using different methodologies: timer interrupts represent 63% of the OS noise. CAOS uses NETTICK to co-ordinate the system activities required for the proper functioning of the HPC system and, at the same time, to reduce the number of timer interrupts, thus, the OS noise. As a side note, the authors in [23] acknowledged that synchronizing the time ticks of the HPC nodes avoids the amplification effect due to the large number of independent nodes. However, they

argued that "synchronizing ticks is analogous to treating the symptom rather than the disease" and proposed to remove the local ticks altogether. As a matter of fact, this cannot be done locally on each node. In fact, the hardware time keeping devices included in the current HPC nodes lack the necessary precision *and* resolution that would allow us to effectively build a tickless OS kernel. In practice, even if we can slow down the rate of the local timer ticks, we must still raise periodic ticks so as to keep track of time elapsing. On the other hand, a real tickless system can be effectively built on top of a global infrastructure like NETTICK.

The other major cause of OS noise is the scheduler: the local kernel can schedule HPC processes out of the CPUs in order to run other (lower priority) processes, including kernel daemons. This problem has also been extensively studied in [8], [11], [19], [23], and several solutions are available [5], [21]. The HPCSCHED scheduler, integrated into CAOS, reduces the OS noise and improves performances [5]; moreover, it includes other useful features for HPC applications such as dynamic hardware resource assignment for automatically and transparently reducing load imbalance. Finally, HPCSCHED is more generic than the one proposed in [21], as it runs on IBM POWER and Intel/AMD architectures.

In [21] the authors described a solution that in some ways resembles CAOS. That work was realized on a Cray cluster that used special *communications processor* to obtain a "common sense of time" among all the nodes. Then, modifying the scheduler, they synchronized all the system activities executed by kernel threads. The main difference with our work is that the synchronization system provided by NETTICK is more general: it can work on any kind of cluster and does not depend on custom hardware.

VIII. CONCLUSIONS AND FUTURE WORK

We have proposed an extension to the Linux kernel aimed at HPC applications in large cluster-based supercomputers. CAOS modifies the standard operating system kernel, in that the kernels running on the cluster nodes activate the most time-consuming system activities only after receiving a command issued by a master node. CAOS has been designed considering HPC applications and their needs.

Though CAOS is still in a prototype phase, it already includes two of its major components: A local task scheduler tailored for HPC applications (HPCSCHED) and a global synchronization mechanism (NETTICK). Thanks to NETTICK, the kernels running on the cluster nodes can rely on a global source of time events. Both components contribute to reduce the OS noise on a single node and thus improve scalability.

Our preliminary results show that, while reducing the OS noise, CAOS imposes no performance impact on HPC applications. On the other hand, CAOS provides significant advantages due to its ability to co-ordinate cluster level operations such as data checkpoint or debugging.

Acknowledgments. The authors gratefully acknowledge support of this research project by the Italian Army Stato Maggiore della Difesa, Distaccamento di Ponte Galeria.

REFERENCES

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

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical report, NASA Advanced Supercomputing (NAS) Division, March 1994.

[3] P. H. 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.

[4] C. Boneti, R. Gioiosa, F. Cazorla, J. Corbalan, J. Labarta, and Mateo Valero. Balancing HPC applications through smart allocation of resources in MT processors. In *to appear in the 22nd IEEE Int. Parallel and Distributed Processing Symp. (IPDPS08)*, Miami, FL, 2008.

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

[6] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Really, 3rd edition, 2005.

[7] K. Correll and N. Barendt. Design considerations for software only implementations of the IEEE 1588 Precision Time Protocol. In *In Conference on IEEE 1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, 2006.

[8] 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.

[9] J.C. Eidson and K. Lee. Sharing a common sense of time. In *IEEE Instrumentation & Measurement Magazine*, 2003.

[10] Fixed Timed Quanta. http://rt.wiki.kernel.org/index.php/FTQ.

[11] 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.

[12] R. Gioiosa, J.C. Sancho, S. Jiang, F. Petrini, and K. Devis. Transparent Incremental Checkpoint at Kernel level: A Foundation for Fault Tolerance for Parallel Computers. *SC—05 (Supercomputing): Int. Conf. for High Performance Computing, Networking, and Storage*, November 2005. http://bravo.ce.uniroma2.it/home/gioiosa/pub/sc05.pdf.

[13] IEEE standard for a precision clock synchronization protocol for networked measurement and control systems. IEEE Std 1588-2008, available at http://ieee1588.nist.gov.

[14] Los Alamos National Laboratory. Btime. http://btime.sourceforge.net/.

[15] J. Loncaric. Btime: a clock synchronization tool for linux clusters. In *Supercomputing Conference SC 2005*, Seattle,WA, 2005.

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

[17] 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 *Supercomputing 07 (SC07)*, 2007.

[18] OpenMP Architecture Review Board. The OpenMP specification for parallel programming. Available at http://www.openmp.org.

[19] 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, November 10–16, 2003.

[20] The precision time protocol daemon (PTPd). http://ptpd.sourceforge.net.

[21] 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.

[22] TOP500 Supercomputer Sites list. http://www.top500.org/.

[23] D. Tsafrir, Y. Etsion, D.G. 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*, pages 303–312, New York, NY, USA, 2005. ACM Press.