A Dynamic Scheduler for Balancing HPC Applications

Carlos Boneti[†], Roberto Gioiosa[*], Francisco J. Cazorla[*], Mateo Valero[*†]

[*]Barcelona Supercomputing Center, Spain

[†]Universitat Politecnica de Catalunya, Spain

{roberto.gioiosa, francisco.cazorla}@bsc.es, {cboneti, mateo}@ac.upc.es

Abstract—Load imbalance cause significant performance degradation in High Performance Computing applications. In our previous work we showed that load imbalance can be alleviated by modern MT processors that provide mechanisms for controlling the allocation of processors internal resources. In that work, we applied static, hand-tuned resource allocations to balance HPC applications, providing improvements for benchmarks and real applications.

In this paper we propose a dynamic process scheduler for the Linux kernel that automatically and transparently balances HPC applications according to their behavior. We tested our new scheduler on an IBM POWER5 machine, which provides a software-controlled prioritization mechanism that allows us to bias the processor resource allocation. Our experiments show that the scheduler reduces the imbalance of HPC applications, achieving results similar to the ones obtained by hand-tuning the applications (up to 16%). Moreover, our solution reduces the application's execution time combining effect of load balance and high responsive scheduling.

I. INTRODUCTION

Modern Supercomputers are often designed with commodity hardware components (for example, Intel or IBM POWER processors) and software. Generally, this kind of Supercomputers are distributed memory machines with a limited number of cores per-node (2-8 cores); the Message Passing Interface (MPI) [2] standard is the most common programming model used in those systems.

HPC applications are, in most of the cases, *Single Program Multiple data* (SPMD), meaning that all processes execute the same code on different data sets. Theoretically, those applications are supposed to reach their synchronization points (e.g. barriers or collective operations) at the same time, exchange data and then continue their tasks. However, several factors cause *load imbalance*. Load imbalance happens, for example, when the amount of input data to be processed by each task in the parallel application is not the same and some tasks take longer than other to reach their synchronization points. In this case we say that the application is *intrinsically* imbalanced. Other factors are external to the application: for example, the Operating System (OS) has also been identified as one of the most important *extrinsic* source of imbalance [9], [22], [24], [28].

The load-imbalance problem is well known and increases with the number of processors in Supercomputers, yet the problem remains open. Several solutions have been proposed in the literature: some of them [1], [25] first analyze the input data and then try to find the best data distribution to reduce application's imbalance. Other solutions [7], [11], instead, balance applications by assigning more computational resources (mainly number of processors) to those processes computing longer.

The arrival of Multi-Threaded (MT) processors[1] providing mechanisms that allow the software to control the processor's internal resource allocation offers new fine-grain ways to solve the problem of HPC application imbalance.

Until recently, software had no control over the resource allocation in MT processors. For example, Chip Multi-Processor (CMP) architectures with a shared cache level implement a cache replacement policy which is not under control of the OS. In this case, the OS composes the workload to run on the cores but cache replacement policy determines which lines have to be evicted from the cache, implicitly deciding how much cache memory allocate to each task and, thus, the speed of the task. This trend has changed with the arrival of the IBM POWER5[TM] [14], [15], [16] and the CELL [12], [13] processors, which allow the software to use, respectively, 8 and 3 levels of *hardware priorities* for each running task. The basic idea is that the higher the hardware priority assigned to a task (with respect to the other task it is co-scheduled with), the higher the amount of processor resources it receives and, hence, the higher its speed. By controlling this hardware prioritization the software can control the speed at which each task runs. In [4] we performed a deep analysis of how the hardware prioritization mechanism of POWER5 processors affects the performance of applications. Two of the main conclusions, also used in this paper, are the following:

1) In general, improving the performance of one task involves a higher performance loss on the task running on the other context, sometimes by an order

---

[1]In this paper we use the term multi-threaded processor to refer to all types of processors executing more than one task at a time: Symmetric Multi-Thread, Chip Multi-Processor, Fine-Grain Multi-threading, Coarse-Grain Multi-threading or any combination of them.

of magnitude. In some cases, in order to reduce the execution time of a task by X% (with respect to the case when both tasks run with the same priority) by increasing its priority, the execution time of the other task in the same core may reduce by more 10X%.

2) Instead of using the full spectrum of priorities (from 0 to 7), we only explore priority differences up to ± 2. Larger priority differences should be used only when the performance of one of the two tasks is not important (e.g., background task).

In [5] we showed, for the first time, how the hardware prioritization mechanism of POWER5 processors can be used to balance HPC applications. In that proof-of-concept paper, we ran a 4-tasks MPI application on a POWER5: in a first test, where we applied the same priority to the two tasks running in a core (default case), we detected which processes, on average, computed the longer and which tasks spent most of their time waiting for incoming messages or on a barrier. In the following experiments we manually increased the priority of the most computing intensive tasks, increasing their speed and reducing the load imbalance. In that paper, the prioritization is applied to processes manually and statically at the beginning of the execution and each process runs with the same priority throughout its execution. With this solution we obtained an improvement of 8% on real HPC applications like SIESTA [26].

In this paper, we propose a dynamic solution implemented as a new task scheduler for Linux 2.6 kernels. The advantages of this new proposal over the static solution are obvious, the most important being that the OS automatically establishes the hardware priority to be assigned to each HPC process with no effort from the user. The second advantage is that the solution is transparent to the user: the only modification in the application source code concerns the scheduling policy (as shown in Section IV). The third advantage is that our scheduler is able to detect the correct hardware priority quickly (in one or two iterations) improving overall performance. Finally, the scheduler is able to catch up with the application in case the application's behavior is dynamic, i.e., not constant throughout the iterations. All these advantages reduce the load imbalance of a HPC application, directly increasing the overall performance.

In order to test our dynamic scheduler, we compared the results we obtained running HPC benchmarks and applications to the results we obtained in [5]. Most interesting is the case of the real application (SIESTA): with our previous static approach we were able to improve the total execution time by 8%; with the solution proposed in this paper, we are able to improve the execution time by almost 6%, combining the effects of the load balancing and the high-responsive task scheduler without any effort from the programmer.

The capability of the IBM POWER5 to allow the software to change processor's internal resource allocation is not something isolated in the design of processors. Several factors support the idea that future supercomputers will use this type of processors. First, nowadays, Multi-Threaded processors are widely used in HPC systems (in addition to many other computing systems like desktops, real-time, etc.) for their good performance/energy consumption and performance/cost ratios. Second, other recent processors like the IBM POWER6$^{\text{TM}}$ [21], provide a similar prioritization mechanism. Third, many computer-architecture researchers advocate that allowing the software to control not only the decode stage of the processor, as it is the case in POWER5 and POWER6, but also other processor shared resources in the chip, like the cache [10], [17], [23], would increase the performance of HPC applications.

The rest of this paper is structured as follows: Section II provides some background on the solutions already proposed in the literature and the capability of the IBM POWER5 to dynamically assign internal resources to each contexts. Section III highlights some of the features of the software designs of the new Linux scheduler framework. Section IV proposes our dynamic task scheduler for balancing HPC applications. Section V shows our experiments on benchmarks and real applications. Finally Section VI provides our conclusions and finalizes the paper.

II. BACKGROUND

A. Related Work

Different solutions have been proposed to solve the load-imbalance problem. Historically, these solutions have been divided into two groups: data distribution and processing distribution.

The first group consists of static and dynamic solutions. Static approaches distribute data using sophisticated tools and achieve good performance results but must be repeated for every application, input data set and architecture. For example, METIS [1] analyzes the application's data set and tries to find the best distribution. Dynamic approaches have also been proposed in the literature: in [25] and [30] the authors try to solve the load-balancing problem of irregular applications by using mesh repartitioning algorithms and evaluating the convenience of repartitioning the mesh or adjust it.

Solutions in the second group, processing redistribution, assign more CPUs to those processes that compute for longer time. Load balancing for openMP applications can be performed using some of the existing loop scheduling algorithms that assign iterations to threads dynamically [3]. When using nested parallelism in openMP, it is possible to assign more threads to those

groups with high loads [7]. In the case of MPI applications, solution for load balancing are more complex, for the number of processes is statically determined when starting the job (in case of malleable jobs), or when compiling the application (in case of rigid jobs) [29].

This problem has also been approached through hybrid programming models, for example combining MPI and openMP. In [11] the authors balance irregular applications by modifying the computational power rather than by using the typical mesh redistribution. In their work the application detects the overloading of some of its processes and creates new threads at run time to alleviate the overloaded CPUs. They observe that one of the problems of their method is that they do not control the operating system decisions, which could reverse theirs.

In this paper we propose a new group of solutions called *processor resource distribution* group where we assign more hardware processor resources to those threads computing the longer. One of the advantages of the proposals in this group is that the *granularity* is smaller than in the other groups, which allows us to be more efficient when reducing the load imbalance. For example, we do not add or remove *processors*, like in the processing re-distribution group, but assign more or less processor internal resources to the most computing intensive tasks. With respect to data distribution group, our solution is transparent to the user and dynamic, which means that the programmer does not have to put any effort for balancing the application and that the solution does not have to be repeated for each application, data input set or architecture.

B. The IBM POWER5 Processor

The IBM POWER5 [14], [15], [16] processor is a dual-core chip where each core is 2-way SMT [19]. Each core has the capability to vary the hardware resources assigned to each thread (or context) at run time by means of a *hardware context priority* (or *hardware thread priority*). The hardware thread priority is an integer value in the range of 0 (the context is off) to 7 (the other context is off and the core is running in Single Thread (ST) mode). The amount of hardware resources assigned to a context increases with the hardware priority value (keeping the other constant).

Each core in the processor prioritizes a task by changing the instruction decode rate, i.e., the number of decode cycles assigned to each context depends on its hardware priority. In general, the higher the priority, the higher the number of decode cycles assigned to the thread and, therefore, the higher the number of shared resources held by the thread.

Let us assume that two tasks (TaskA and TaskB) are running on a POWER5 core with priorities PrioA and PrioB, respectively. Every time slice of R cycles the task

TABLE I
DECODE CYCLES ASSIGNED TO TASKS BASED ON THEIR PRIORITIES

Priority difference	R	Decode cycles (A)	Decode cycles (B)
0	2	1	1
1	4	3	1
2	8	7	1
3	16	15	1
4	32	31	1
5	64	63	1

TABLE II
PRIVILEGE LEVEL AND OPERATION TO SET EACH PRIORITY LEVEL

Priority	Priority level	Privilege level	or-nop instruction
0	Thread off	Hypervisor	-
1	Very low	Supervisor	or 31,31,31
2	Low	User	or 1,1,1
3	Medium-Low	User	or 6,6,6
4	Medium	User	or 2,2,2
5	Medium-high	Supervisor	or 5,5,5
6	High	Supervisor	or 3,3,3
7	Very high	Hypervisor	or 7,7,7

with the lower priority receives 1 decode cycle while the task with the higher priority receives $(R-1)$ cycles. R is computed as:

$$R = 2^{|PrioA-PrioB|+1}$$

Table I shows how R is computed according to the priority difference between TaskA and TaskB (PrioA-PrioB) and how many decode cycles are assigned to each task. For instance, assuming that the hardware priority of TaskA is 6 and the hardware priority of TaskB is 2 (the difference is 4), the core fetches 31 times from TaskA and once from TaskB (more details on the hardware implementation are provided in [8]). It is clear that the performance of TaskA should increase to the detriment of TaskB.

If the hardware thread priority of a context is 0, 1 or 7, the behavior of the hardware prioritization mechanism does not follow Table I [14], [15], [16]. Priority 0 means that the thread is switched off; priority 7 means the thread is running in ST mode (i.e., the other thread is off) and priority 1 means that the context is running a "background" thread assigning it all the hardware resources left over by the "foreground" thread running on the other context.

Hardware priorities in the IBM POWER5 can be changed by issuing an or-nop instruction. In order to change its thread priority, a task has to execute an instruction like or X,X,X, where X is a specific register number (see Table II). This instruction does not perform any operation except changing the hardware thread priority of the task. Table II shows the priorities, the privilege level required to set each priority and how to change

priority using this interface. The OS (supervisor) can set 6 out of 8 priority values, from 1 to 6; user software can only set priority 2, 3, 4; the Hypervisor can always span the whole range of priority values.

III. THE LINUX SCHEDULER FRAMEWORK

A new process task scheduler (the *Complete Fair Scheduler*, CFS) has been introduced in the Linux kernel version 2.6.23. This new scheduler replaces the old *O(1)* [6] scheduler used in Linux 2.6 for several years. The *O(1)* scheduler provided good performance and its overhead was constant regardless of the number of runnable processes. However, this scheduler was not free of problems, such as consuming too much memory even with few runnable tasks. The CFS aims to solving some of those problems.

Together with the new CFS algorithm, a new *scheduler framework* has also been introduced, mainly to simplify the structure of the task scheduler. The new framework divides the scheduler in two main components: three *Scheduling Classes*, which implement the policy details, and a *Scheduler Core*, which handles the Scheduling Classes as *objects*, i.e., calling the appropriate Scheduling Classes methods for any low-level operations (for example, selecting the next task to run or accounting for the time elapsed). Each of the three Scheduling Classes contains one ore more scheduling policies (see Figure 1(a)).

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, previously called SCHED_OTHER, and SCHED_BATCH); finally, the last class contains the idle process (SCHED_IDLE).

The order with which the Scheduling Classes are linked together 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. For example, no processes from the CFS class will be selected if there is one process in the real-time class; this design choice preserves the semantic of the SCHED_FIFO and SCHED_RR policies. In the same way, the idle process will never be selected if there are runnable processes in one of the other classes.

When the scheduler is invoked, the Scheduler Core starts looking for the best process to run from the highest priority class (i.e., the real-time class) and checks whether there are runnable processes in this class. If the class contains at least one process, the scheduler selects this process and assigns it to the CPU. If the class

is empty, i.e., no runnable process available, then the Scheduler Core moves to the next class. This operation repeats until the Core Scheduler finds a runnable task to run on the CPU. Notice that the idle class always contains at least the idle process, thus the scheduler cannot fail in its search.

A very interesting property of the new scheduler framework is that each class may provide different data structures and algorithms to select the next process to run. For example, the real-time class uses a set of priority, round-robin run queue lists, one list for each real time priority (0-99). The real-time scheduler first selects the highest (non-empty) priority run queue and then picks up the first task in the list. In fact, a real-time task is either SCHED_FIFO, in which case the task stays in the first position until it yields the CPU, or a SCHED_RR, in which case the process is moved to the back of the queue if its time slice expires. This algorithm is essentially the old *O(1)* scheduler algorithm and maintains the *O(1)* scheduler's implementation details (like the 0-cost swap between the active and expired arrays).

The CFS class, instead, uses a red-black tree and does not use the concept of time quantum. Each process receives a time slice proportional to the actual workload (the higher the number of running processes, the smaller the time slice). The key concept is the time spent by a runnable task waiting for a CPU (i.e., waiting to be executed). This value is used to sort the tasks in the red-black tree so that the "leftmost task" in the tree is the process that has been waiting for more time (i.e., the one with gravest need to run), therefore the next task to run. The CFS scheduler tries to balance the execution of the runnable tasks so that no one waits for a CPU more than a maximum allowed amount of time[2] (*latency*). As the time passes, the waiting time of the running process is decreased at every timer interrupt (or scheduling event) by the amount of time the task has been running (minus its fair running time). As the waiting time of the running task decreases, the task may eventually be moved to the right side of the red-black tree. Sooner or later the running task will not be the "leftmost task" anymore, in that moment the CFS scheduler will select another task.

As the previous examples show, the Scheduling Classes may have completely different algorithm and data structures. As a matter of fact, the new scheduler framework allows kernel developers to write scheduler algorithms specifically tailored for a class of applications. Moreover, adding a new scheduler algorithm is easier than in the past and does not require heavy modification of pre-existing kernel code.

---

[2]The default maximum value for normal tasks is 20ms.

(a) Standard Linux Scheduling Classes
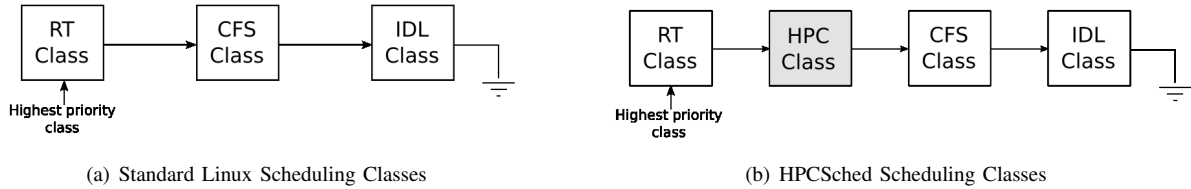
(b) HPCSched Scheduling Classes

Fig. 1. Scheduling classes for the standard and the modified Linux kernel

IV. THE HPC SCHEDULER

In this paper we propose a dynamic mechanism to balance MPI applications using the hardware priority mechanism provided by IBM POWER5 processors. We implemented our dynamic solution inside the Linux kernel as a new scheduler (*HPCSched*) for a special class of applications (HPC applications).

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.

Since we want to prioritize HPC over normal processes, we introduced the HPCSched class between the Real-Time and the CFS class (see Figure 1(b)). In this way, we preserve the semantic of the real-time tasks (`SCHED_FIFO` and `SCHED_RR`) and give a higher priority to HPC processes over normal tasks.

The HPC scheduler we propose 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 new hardware priority for the task.

Mechanism: Architecture-dependent, utility functions necessary to set the new hardware priority or read the current priority of a task.

A. Scheduling policy

Taking advantage of the new scheduler framework described in the last Section, we introduced a new Scheduler Class (`sched_hpc`) and a new scheduler policy (`SCHED_HPC`) for HPC applications. A user can move an application to the HPC class by means of the standard `sched_setscheduler()` system call. Actually, this is all the effort the user has to put in order to use our new scheduler (comparable to the use of the `nice()` system call commonly used in HPC applications).

Our scheduler algorithm is specific for HPC applications, more specifically for MPI applications. The typical way of running MPI applications on current supercomputers is to run one MPI process per-CPU.

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 worth to have a complex algorithm for selecting the next task to run. In fact, with 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 performing (for example, the scheduler does not have to balance any tree). Nevertheless, we implemented two algorithms:

FIFO: Fist-In-First-Out algorithm. The selected task will run until the end or until it yields the CPU.
RR: Round-Robin algorithm. Each task has a pre-defined time slice. When this time slice expires, the task is placed at the end of the run queue.

We observed that, with one process per CPU running at any given moment, there is essentially no difference between these two policies, thus, we only include the results for the round robin policy in this paper. However, as we have already remarked, the scheduling policy is independent of the other components, hence, it can be changed, if required, without affecting the heuristics or the applying mechanism.

In the new Linux kernel framework, workload balancing, i.e., splitting evenly the workload among all the available domains [6] (at core-, chip- and system-level), is also performed at Scheduling Class level. Every Scheduling Class has its own workload balancing algorithm, which means that each CPU has, roughly, the same number of real-time or normal tasks. As a side effect, each CPU runs, more or less, the same number of 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, busiest run queue lists to its run queue.

We implemented our HPC workload balancing algorithm making each processor domain [6] running the same number of processes. For example, in a POWER5 system there are three domain levels: chip level, core level and context level (a context 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 less tasks than another core will try to pull tasks from the other core. For example, if one

core of an IBM POWER5 processor (a domain composed by two contexts) contains one HPC task and the second core contains three tasks, the first core will try to pull one HPC task from the second core so that each core domain contains two processes so to make the workload balanced.

B. Load Imbalance Detector and Heuristics

MPI applications alternate *computing phase* (when a process is runnable) with *waiting phases* (when a process is waiting for an incoming message or for synchronization, thus, not runnable). We consider the sum of a computing phase and of a waiting phase as one *iteration* of the MPI application.

In some HPC applications during each iteration all the tasks perform the same operations (most of the time on the same amount of data), with an iterative structure.

Our solution learns from the execution history of a process: the general idea is that if a task does not have a high CPU utilization during the 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 than try to reduce the error in they made in the approximation. The Load Imbalance Detector assumes that the iteration i is representative of the iteration $i + 1$, hence, the HPC scheduler can change the task's priority and apply the new priority before the iteration $i + 1$ starts. The goodness of our solution strongly depends on how close this guessing is to the optimum solution. If the guessing is not correct, in the iteration $i + 1$ the application may result to be even more imbalanced than in the iteration i. Hopefully, the scheduler will detect this anomaly during the iteration $i + 1$ and apply the right priority in the iteration $i + 2$.

Clearly, not all the applications present a well defined iterative structure with a barrier at the end of the iterations. Some applications, like SIESTA, are more dynamic or do not require all the processes to be synchronized with a global barrier. If the iteration i is not representative of the iteration $i + 1$, our current heuristics will probably fail to balance the application and new heuristics are required. We leave the study of new heuristics for future work.

The scheduler may require some iteration 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. Sometimes it is not possible to balance an application, for example because the hardware priority mechanism of the POWER5 processor is too coarse grain. In this case the scheduler will oscillate between two solutions without being able to find the perfect balance, hopefully still reducing the overall load imbalance.

The problem here is to find the correct trade-off between *performance* (computing the next priority quickly), *responsiveness* (converging to the correct priority in few iterations) and *adaptability* (changing the priority whenever the tasks' behavior changes).

In order to compute the next task priority quickly our heuristics are based on the CPU utilization of a process, a simple metric that does not require complex computations. Ideally, the scheduler should look at the tasks running on the two contexts of a POWER5 core simultaneously and then compute the correct priority for the current task. In fact, the performance of the current task depends on the difference between its priority and the priority of the task running on the other context. However, this would require to acquire a lock on the other context's run queue (in order to ensure that no process switch occurs), thus, stalling the other context until the new priority has been computed. Things become even more complex as the HPC scheduler needs to be sure that the process running on the other context is a SCHED_HPC tasks, for the lock on the task descriptor should also be acquired (in order to avoid concurrent access to the task descriptor). This solution could be quite expensive in terms of performance (though very precise). Hence, we decided to implement a simpler solution that only computes the new priority of a HPC task according to its statistics (thus, not considering the task running on the other context).
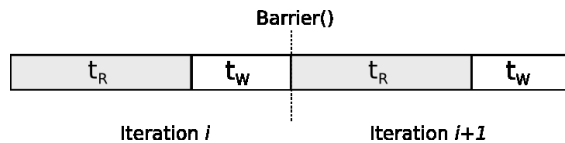

Fig. 2. HPC application iterative behavior

While a task is running, the scheduler collects several metrics, such as the tasks' 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 the 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.

From our study in [4], we learned that priority differences greater than 2 drastically reduce the performance of the low priority task. Therefore, we limited the range of priorities that the HPC scheduler explores to

$[4, 6]$ (where 4 is the normal priority assigned to each task at the beginning), so that the maximum allowed priority difference is ± 2. In this way, the performance of the highest priority task might increase up to 95% of the maximum performance improvement but the lower priority task's performance does not decrease too much.

Once the information about the tasks' progress have been stored, the HPC scheduler has to decide whether to increase, decrease or keep the same priority for the current process in the next iteration. Since HPC applications can be very different, it is hard to find an heuristic that works well in all the cases. In this paper, we implemented and tested two heuristics: the first heuristic (*Uniform* heuristic) targets constant applications, i.e., applications that do not change drastically their behavior from one iteration to another. The second heuristic (*Adaptive* heuristic) is more aggressive and tries do adapt to different program phases. Which heuristic is better for a specific application depends on the characteristics of the applications itself. Section V shows how an application takes more advantages from one heuristic than from the other. We decided to allow the user to select which heuristic to use when compiling the kernel. Once the heuristic has been chosen, the user can set some parameters at run time to tune the heuristic and make it more suitable for the application.

Uniform prioritization: This heuristic uses the global utilization ratio of a task. Every scheduling tick, the OS accumulates the running time for the active task and updates its utilization; the sleeping time is accounted when a task wakes up at the beginning of the new iteration. Just before starting the new iteration, the Load Balancer Detector checks the application's imbalance and the heuristic eventually applies the new task priority according to the global utilization,

We introduced two configurable limits, `LOW_UTIL` and `HIGH_UTIL` that define the boundaries when a task is considered to be a low, medium or high utilization task. Those boundaries are required to avoid that the scheduler changes too quickly the priority of a task, oscillating between two possible solutions. For the experiments presented in Section V, we set `HIGH_UTIL` to 85 and `LOW_UTIL` to 65. The heuristic can be tuned by the user through specific entries in the *sysfs* filesystem.

The Uniform heuristic is very simple and adds negligible overhead to the task scheduler. The heuristic properly balance applications with constant behavior although it could be slow to adapt to different behaviors of the program. If the heuristic is able to balance the application, i.e., to find a *stable state*, the Load Imbalance Detector only checks whether the application maintain the same behavior or not, without changing the priority of each task. If the application's behavior changes, the Load Imbalance Detector tracks this and the heuristic

selects the right priority for the next iterations.

Adaptive prioritization: The *Uniform* heuristic may be too slow to adapt to new scenario if the application changes its behavior quickly, especially if the application runs for a long time (in which case it is hard to impact the global utilization, as Section V-B shows. We implemented another heuristic, that we called *Uniform*, which gives more weight to the recent history of the application. With this heuristic, the task utilization in the $i - th$ iteration is computed as $U_i = G * U_g(i-1) + L * U_l(i)$, where $U_g(i-1)$ is the global utilization until the iteration $i - 1$ and $U_l(i)$ is the CPU utilization of the last iteration i. G and L (with $G + L = 1$) weight, respectively, the global and the last utilization. These parameters can be used to make the heuristic more or less aggressive: in fact, an aggressive heuristic (for example, $L = 0.90$ and $G = 0.10$) quickly adapts to the application's behavior but may over-react, meaning that even small changes caused by external factors, like the OS noise, may cause the heuristic to change the task priority. On the other hand, if the value of G is close to 1, the *Adaptive* heuristic behaves like the *Uniform* heuristic.

As for the Uniform heuristic, the Adaptive heuristic can also be tuned at run time using different values for `HIGH_UTIL`, `MAX_PRIO` (the maximum allowed priority) and `MIN_PRIO`. Moreover, if the Load Balancer stops to change the tasks' priority if it detects that the application is well balanced.

C. Mechanism

This is the only architecture-depended part of our solution. In fact, while the HPC scheduler can be used on any architecture and may, eventually, provide some performance improvement (because the HPC class has higher priority than the CFS class), balancing an MPI application assigning more or less hardware resources to a process can only be done if the underneath processor supports this feature.

V. Experiments

In this section we evaluate the performance of our HPC scheduler and compare it to the standard CFS scheduler and the static solution proposed in [5]. As we said in Section IV, the goodness of the HPC scheduler strongly depends on the heuristics we apply. For this reason, some application may benefit more than other from an heuristic while other may experiment some performance degradation.

Like in [5], we present our results for three different cases: MetBench, our micro-benchmark suite (Section V-A), BT-MZ from the NAS benchmark suite (Section V-C) and SIESTA, a real application (Section V-D). In order to evaluate how our HPC scheduler handles

dynamic applications, in this paper we also present results for MetBenchVar V-B, a version of MetBench that changes its behavior after k iterations, reversing the load imbalance.

We performed the experiments on an IBM OpenPower 710 server, equipped with one POWER5 processor. We ran our experiments on a standard Linux 2.6.24 (the last available Linux kernel at the moment of writing this paper) and our modified Linux kernel. All the benchmarks are MPI applications (in the experiments we used the MPI-CH 1.0.4p1 implementation of MPI protocol). In order to graphically show how HPCSched balances an MPI application, we used PARAVER [20], a visualization and performance analysis tool developed at CEPBA to collect data and statistics and to show the trace of each process during the tests.

As a performance metric we use CPU utilization of each task, and the total execution time of the application. Reducing the load imbalance lead to higher CPU utilization but does not necessarily improve performance: other factors, like the communication pattern of the application, may play an important role and reduce the performance of the application. On the other hand, HPCSChed is also able to improve the performance of an application reducing the overhead an application running with the standard CFS scheduler may suffer.

A. MetBench

MetBench (Minimum Execution Time Benchmark) is a suite of MPI micro-benchmarks developed at BSC which structure is representative of the real applications running on MareNostrum. MetBench consists of a *framework* and several *loads*. The framework is composed by a *master* process and several *workers*: each worker executes its assigned load and then waits for all the others to complete their task. The master maintains a strict synchronization between the workers: once all the workers have finished their tasks, the master eventually starts another iteration. The master and the workers only exchange data during the initialization phase and use an `mpi_barrier()` to get synchronized.

In this experiment we introduce imbalance in the MPI application by assigning to a worker a larger load than the worker running on the same core. In this way, the faster worker will spend most of its time waiting for the slower worker to process its load. Figure 3(a) shows part of the execution trace of our reference case, where MetBench runs with the default CFS (Completely Fair Scheduler). In this figure dark gray is the computing time, while light gray is the waiting or communication time. Table III shows that two of the MetBench workers are idle for about 75% of the time. Figure 3(b) shows the solution proposed in [5], where we were able to statically balance the application: the execution time decrease

TABLE III
METBENCH BALANCED AND IMBALANCED CHARACTERIZATION

Test	Proc	% Comp	Priority	Exec. Time
Baseline 2.6.24	P1	25.34	4	81.78s
	P2	99.98	4	
	P3	25.32	4	
	P4	99.97	4	
Static	P1	99.97	4	70.90s
	P2	99.64	6	
	P3	99.95	4	
	P4	99.64	6	
Uniform	P1	96.17	-	71.74s
	P2	98.57	-	
	P3	90.94	-	
	P4	99.57	-	
Adaptive	P1	80.64	-	71.65s
	P2	99.52	-	
	P3	87.52	-	
	P4	99.20	-	

from $74.64sec$ to $70.90sec$, with an improvement of about 13%. The static approach we used in [5] require previous knowledge of the application and effort from the programmer to detect the load imbalance and to properly assign hardware resources to each task.

Figures 3(c) and 3(d) show how HPCSched is able to properly balance MetBench after the first iteration. In fact, the behavior of MetBench is constant, thus, each iteration is representative of the following ones. In Figure 3(c), the Load Imbalance Detector detects the imbalance in the first iteration [3] and the *Uniform* heuristic computes and apply the correct priority for each task before the beginning of the second iteration. At the end of the second iteration, the Load Imbalance Detector detects no imbalance, thus there is no need of trying to balance again the application. The execution time with the *Uniform* heuristic is $71.74sec$ (about 12%) of improvement), comparable with the static solution shown in Figure 3(b) but without any effort from the programmer.

The *Adaptive* heuristic also provides good performance: the total execution time is $71.65sec$ (about 12% of improvement). In this experiment the *Adaptive* heuristic uses a very aggressive approach (10% global history, 90% last iteration), thus, even a small variation (caused, for example, by OS noise) may stimulate the heuristic to change the priority of some task. If this happens, like in Figures 3(d), the heuristic may respond too quickly and take the wrong decision. However, Figures 3(d) also shows how the *Adaptive* heuristic is able to recover after the error.

---

[3]Notice that the first iteration already uses non standard priority: this is the result of the initialization phase, not visible in the trace

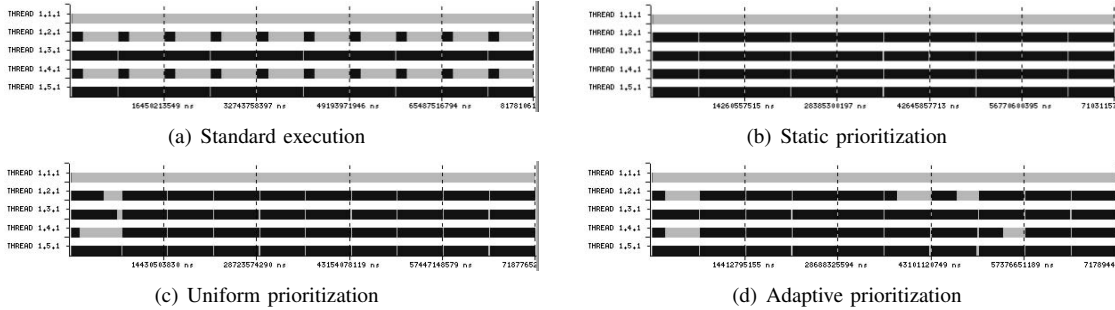(a) Standard execution

(b) Static prioritization

(c) Uniform prioritization

(d) Adaptive prioritization

Fig. 3. Effect of the proposed solution on MetBench.

(a) Standard execution

(b) Static prioritization

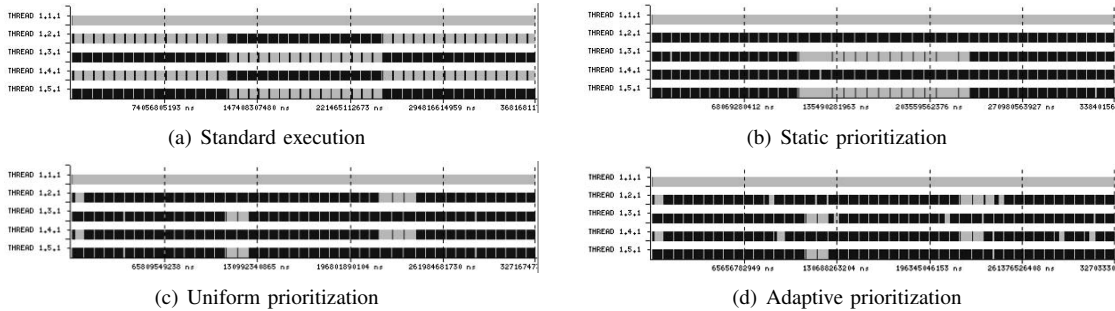(c) Uniform prioritization

(d) Adaptive prioritization

Fig. 4. Effect of the proposed solution on MetBenchVar.

B. MetBenchVar

MetBenchVar is a slightly modified version of Met-Bench where the workers change their behavior after k iteration. Figure 4(a) shows the standard execution of MetBenchVar with $k = 15$: at the beginning $P1$ and $P3$ execute a small load while $P2$ and $P4$ a large load. At the 15th iteration, $P1$ and $P3$ start to execute the large load while $P2$ and $P4$ perform their task on the small load. In this way, we reverse the load imbalance at run time making the application's behavior dynamic. At the 30th iteration, we switch again the behavior of the tasks. Figure 4(b) shows how a static works in this case: the application is perfectly balanced in the first (iterations 1-15) and third period (iteration 31-45) but the prioritization is reversed in the second period (iterations 16-30), as a result, in the second period the application performs worst than in the standard case.

Our dynamic solution, instead, is able to detect that the application's behavior has changed and dynamically adjust the priority of each task in order to re-balance the application. Figure 4(c) shows how HPCSched performs in this experiment when applying the *Uniform* heuristic: after the switching in the 15th iteration, the scheduler needs two more iterations to detect and correct the new load imbalance. However, after the second switch, the scheduler needs three more iterations to detect and correct the load imbalance and the trend continue if the application runs for longer time. Since the *Uniform* heuristic uses the global history to detect the imbalance,

it is expected that the longer the application runs, the less responsive is the scheduler. Thus, increasing the value of k or the number of periods makes the scheduler slower to adapt to the new scenario. As Table IV, the execution time reduces from $368.17sec$ to $327.17sec$, with an improvement of about 11%).

Figure 4(d) shows how the *Adaptive* heuristic preforms in this experiment: with $k = 15$, the scheduler always needs only two iterations to detect and correct the load imbalance but, as for the previous case, some times the heuristic is too aggressive and respond too quickly. Again, the *Adaptive* heuristic is able to correct its over-reaction in the following iteration and to reduce the execution time to $326.41sec$ (about 11% of improvement).

C. BT-MZ

Block Tri-diagonal (BT) is one of the NAS Parallel Benchmarks (NPB) suite. BT solves discretized versions of the unsteady, compressible Navier-Stokes equations in three spatial dimensions, operating on a structured discretization mesh. BT Multi-Zone (BT-MZ) [18] is a variation of the BT benchmark which uses several mesh (named *zone*) for, in realistic applications, a single mesh is not enough to describe a complex domain.

Besides the complexity of the algorithm, BT-MZ shows a behavior very similar to MetBench: every process in the MPI application performs some computation on its part of the data set and then exchanges data with its neighbors asynchronously (using `mpi_isend()`

(a) Baseline execution

(b) Static prioritization

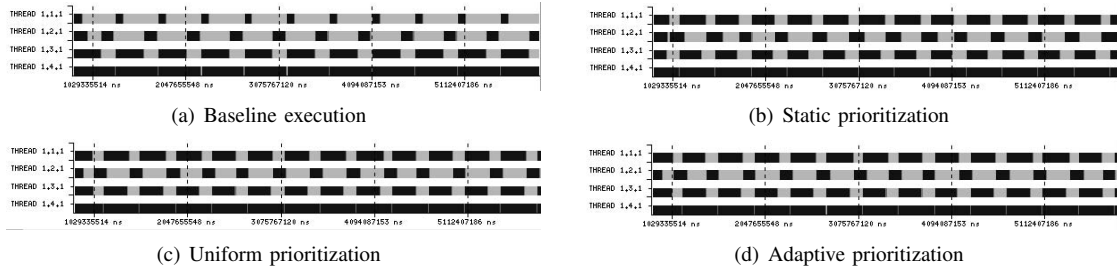(c) Uniform prioritization

(d) Adaptive prioritization

Fig. 5. Effect of the proposed solution on BT-MZ. Each trace represents only some iterations of the application.

TABLE IV
VARIABLE-METBENCH BALANCED AND IMBALANCED
CHARACTERIZATION

Test	Proc	% Comp	Priority	Exec. Time
Baseline 2.6.24	P1	50.24	4	368.17s
	P2	75.09	4	
	P3	50.22	4	
	P4	75.08	4	
Static	P1	99.97	4	338.40s
	P2	68.06	6	
	P3	99.94	4	
	P4	68.04	6	
Uniform	P1	91.47	-	327.17s
	P2	95.55	-	
	P3	91.44	-	
	P4	95.33	-	
Adaptive	P1	89.61	-	326.41s
	P2	93.08	-	
	P3	89.99	-	
	P4	95.15	-	

TABLE V
BT-MZ BALANCED AND IMBALANCED CHARACTERIZATION

Test	Proc	% Comp	Priority	Exec. Time
Baseline 2.6.24	P1	17.63	4	94.97s
	P2	29.85	4	
	P3	66.09	4	
	P4	99.85	4	
Static	P1	70.64	4	79.63s
	P2	42.22	4	
	P3	60.96	5	
	P4	99.85	6	
Uniform	P1	70.31	-	79.81s
	P2	37.18	-	
	P3	65.29	-	
	P4	99.85	-	
Adaptive	P1	70.31	-	79.92
	P2	37.30	-	
	P3	65.30	-	
	P4	99.83	-	

and `mpi_irecv()`). After this communication phase (0.10% of the total execution time) each process waits (with a `mpi_waitall()` function) for its neighbors to complete their communication phases. In this way, each process gets synchronized with its neighbors (note that this does not mean that each process gets synchronized with all the other processes). Once a process has exchanged all the necessary data, a new iteration can start and the previous behavior repeats again until the end of the application (in our experiments we used BT-MZ with default values: class A with 200 iterations).

Figure 5 shows how HPCSched is able to balance BT-MZ achieving results similar to the static prioritization (Figure 5(b)). Both the *Uniform* (Figure 5(c)) and the *Adaptive* (Figure 5(d)) heuristics are able to balance the application and remain in the stable state. Table V shows that the performance improvement is about 16% for both heuristics over the standard case shown in Figure 5(a)

D. Siesta

SIESTA [27] is a method for *ab initio order-N materials simulation*, specifically it is a self-consistent density functional method that uses standard norm-conserving pseudo-potentials and a flexible, numerical linear combination of atomic orbitals basis set, which includes multiple-zeta and polarization orbitals.

In this experiment we used the *benzene* particle as input set and we noticed that the application presents an imbalance caused by both the algorithm and the input set (see Figure 6(a) and Table VI). SIESTA behavior, however, is not constant during each iteration, as can be seen in Figure 6(a) and an iteration is not necessarily representative of the next one; this variability decreased the effectiveness of our static balancing.

As can been seen in Table VI, both the *Uniform* and the *Adaptive* heuristics are only able to reduce the load imbalance marginally (the CPU utilization of each task slightly increases). However, the HPCSched is able to improve the application's performance, reducing the total execution time from $81.49sec$ to $76.82sec$ for the *Uniform* heuristic and $76.91sec$ for the *Adaptive* heuristic. In both cases the improvement is about 6%.

Clearly this improvement does not come from load imbalance reduction but from the other components of our solution, in this case, from the scheduler policy. Fig-
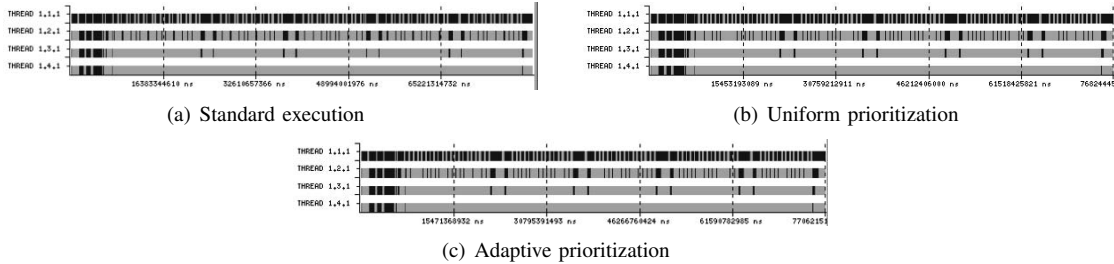
(a) Standard execution

(b) Uniform prioritization

(c) Adaptive prioritization

Fig. 6. Effect of the proposed solution on Siesta.

ure 6(a) shows that the execution phases are very small and that the tasks need to exchange several messages. While waiting for an incoming message, tasks sleep and need to be waken up as soon as the message arrives. The time between the arrival of the message and the moment the task resumes its execution is called *scheduler latency*: SIESTA is very sensible to this kind of *OS noise*. With the CFS scheduler, whenever a task becomes runnable, it has to compete with all the other processes in the system for the CPU. An SCHED_HPC task that wakes up, instead, has to compete only with the other tasks in its class: considering our initial assumption (i.e., usually only one HPC task per-CPU at any given time) the task is able to immediately run on the CPU, thus, its scheduling latency is reduced.

TABLE VI
SIESTA BALANCED AND IMBALANCED CHARACTERIZATION

Test	Proc	% Comp	Priority	Exec. Time
Baseline	P1	98.90	4	81.49s
2.6.24	P2	52.79	4	
	P3	28.45	4	
	P4	19.99	4	
Uniform	P1	98.81	-	76.82s
	P2	53.38	-	
	P3	31.41	-	
	P4	21.68	-	
Adaptive	P1	98.81	-	76.91s
	P2	53.40	-	
	P3	31.47	-	
	P4	21.71	-	

VI. CONCLUSIONS AND FUTURE WORK

HPC applications are, in most of the cases, *Single Program Multiple data* (SPMD), meaning that all processes execute the same code on different data sets. Because of load imbalance these applications do not reach their synchronization points at the same moment, as they are supposed to do.

In [5] we showed how assigning more hardware resources to the most intensive task in an MPI application can reduce the load imbalance and improve performance. We performed this study with a static, hand-tuned approach. In this paper we proposed a new dynamic solution for balancing HPC application, HPC-Sched. We implemented our solution as a new task scheduler for Linux 2.6 kernels composed by three components: the scheduling policy (SCHED_HPC), the metrics and heuristics (Uniform and Adaptive) and the hardware mechanism.

The heuristic used to balance the tasks in the parallel application is critical to achieve good results: in this paper we showed that the perfect heuristic depends on the application's characteristics and that constant applications may not react very well with an aggressive, high-responsiveness heuristic and vice-versa.

We tested our new Linux scheduler on an IBM POWER5 machine using four different applications: MetBench, a suit of micro-benchmarks, MetBenchVar (which performs like MetBench but with different periods of execution), BT-MZ, from the NAS benchmarks suite, and SIESTA, a real application. The results we obtained are good, though they depend on the used heuristic. Our solution works well for constant application like MetBench or BT-MZ providing good results (12% and 16% of performance improvement, respectively). For applications that changes their behavior at run time, HPCSched achieve good performance compared with what a programmer can manually do: MetBench-Var shows a performance improvement of 11% while SIESTA an improvement of about 6%. Our previous static approach we could improve the overall execution time by 8% but that solution required the programmer to manually balance the application while HPCSched is able to balance the application automatically.

We also showed that the improvement comes from a combination of two factors: the scheduling policy and the load balancing.

As future work we plan to expand our solution at cluster level: in fact, HPCSched is a task scheduler able to balance HPC application inside a node but modern Supercomputers consists of Thousands of nodes. In this case there is another level of load balancing which consists of assigning the correct group of tasks to each node

(*gang scheduling*) considering that the local scheduler (in our case HPCSched) is able to dynamically assign more or less hardware resource to each task. Moreover, we would like to find an heuristic capable of performing well (even if not optimal) for both constant and dynamic applications.

ACKNOWLEDGMENT

REFERENCES

[1] Metis - family of multilevel partitioning algorithms. http://www.cs.umn.edu/ metis.

[2] The Message Passing Interface (MPI) Standard. http://www-unix.mcs.anl.gov/mpi/.

[3] E. Ayguade, B. Blainey, A. Duran, J. Labarta, F. Martnez, X. Martorell, and R. Silvera. Is the schedule clause really necessary in openmp? In *In Proceedings of the International Workshop of OpenMP Applications and Tools , Lecture Notes in Computer Science. Toronto, Canada.*, pages 147–159, Jun 2003.

[4] C. Boneti, F. Cazorla, R. Gioiosa, C-Y. Cher, A. Buyuktosunoglu, and M. Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *The 35th International Symposium on Computer Architecture (ISCA).*, Beijing, China, June 2008.

[5] 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 *International Parallel and Distributed Processing Symposium (IPDPS).*, Miami, USA, April 2008.

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

[7] A. Duran, M. Gonzalez, J. Corbalan, X. Martorell, E.Ayguade, J. Labarta, and R. Silvera. Automatic thread distribution for nested parallelism in openmp. In *.International Conference on Supercomputing (ICS05).In Proceedings of the 19th ACM International Conference on Supercomputing. Cambridge, Massachusetts, USA*, pages 121–130, June 2005.

[8] B. Gibbs, B. Atyam, F. Berres, B. Blanchard, L. Castillo, P. Coelho, N. Guerin, L. Liu, C. Diniz Maciel, and C. Thirumalai. *Advanced POWER Virtualization on IBM eServer p5 Servers: Architecture and Performance Considerations*. IBM Redbook, 2005.

[9] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of System Overhead on Parallel Computers. In *The 4th IEEE International Symposium on Signal Processing and Information Technolog y (ISSPIT 2004)*, Rome, Italy, December 2004. Available from http://bravo.ce.uniroma2.it/home/gioiosa/pub/isspit04.pdf.

[10] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, 2007.

[11] W. Huang and D. Tafti. A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications. In *Proceedings of Parallel Computational Fluid Dynamics 99.*

[12] IBM. Cell broadband engine architecture.

[13] IBM. Cell broadband engine programming handbook.

[14] IBM. PowerPC Architecture book: Book I: User Instruction Set Architecture.

[15] IBM. PowerPC Architecture book: Book II: PowerPC Virtual Environment Architecture.

[16] IBM. PowerPC Architecture book: Book III: PowerPC Operating Environment Architecture.

[17] Ravi R. Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Donald Newell, Yan Solihin, Lisa R. Hsu, and Steven K. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, pages 25–36, 2007.

[18] H. Jin and R.F. Van der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. *J. Parallel Distrib. Comput.*, 66(5):674–685, 2006.

[19] R. Kalla, B. Sinharoy, and J.M. Tendler. IBM POWER5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24:40–47, 2004.

[20] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A parallel program development environment. In *Euro-Par, Vol. II*, pages 665–674, 1996.

[21] H.Q. Le, W.J. Starke, J.S. Fields, F.P. O'Connell, D.Q. Nguyen, B.J. Ronchetti, W.M. Sauer, E.M. Schwarz, and M. T. Vaden. IBM power6 microarchitecture. *IBM Journal of Research and Development*, 51:639–662, 2007.

[22] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman. The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *International Conference for High Performance Computing, Networking, Storage and Analysis SC07*, November 2007.

[23] Kyle J. Nesbit, Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and J. E. Smith. A framework for managing multicore resources. *To appear in IEEE Micro special issue on Interaction of Computer Architecture and Operating System in the Many-core Era. Available at http://www.ece.wisc.edu/~nesbit/papers/VPM_ieee_micro08_draft.pdf*, 2008.

[24] 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.

[25] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. Technical report.

[26] Siesta-Project. Siesta: A linear-scaling density-functional method. http://www.uam.es/siesta/.

[27] J.M. Soler, E. Artacho, J.D. Gale, A. Garca, J. Junquera, P. Ordejn, and D. Snchez-Portal. The siesta method for ab initio order-n materials simulation. *Journal of Physics: Condensed Matter*, 14(11), 2002.

[28] D. Tsafrir, Y. Etsion, D.G. Feitelson, and S. Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM Press.

[29] G. Utrera, J. Corbaln, and J. Labarta. Implementing malleability on mpi jobs. In *The 13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004)*, pages 215–224, October 2004.

[30] C. Walshaw and M. Cross. Dynamic mesh partitioning and load-balancing for parallel computational mechanics codes. 2002.