Balancing HPC Applications Through Smart Allocation of Resources in MT Processors

Carlos Boneti†, Roberto Gioiosa*, Francisco J. Cazorla*, Julita Corbalan*†,
Jesus Labarta*†, Mateo Valero*†
*Barcelona Supercomputing Center, Spain
†Universitat Politecnica de Catalunya, Spain
{roberto.gioiosa, francisco.cazorla}@bsc.es, {cboneti, julita, jesus, mateo}@ac.upc.es

***Abstract*—Many studies have shown that load imbalancing causes significant performance degradation in High Performance Computing (HPC) applications. Nowadays, Multi-Threaded (MT[1]) processors are widely used in HPC for their good performance/energy consumption and performance/cost ratios achieved sharing internal resources, like the instruction window or the physical register. Some of these processors provide the software hardware mechanisms for controlling the allocation of processor's internal resources. In this paper, we show, for the first time, that by appropriately using these mechanisms, we are able to control the tasks speed, reducing the imbalance in parallel applications transparently to the user and, hence, reducing the total execution time. Our results show that our proposal leads to a performance improvement up to 18% for one of the NAS benchmark. For a real HPC application (much more dynamic than the benchmark) the performance improvement is 8.1%. Our results also show that, if resource allocation is not used properly, the imbalance of applications is worsened causing performance loss.**

I. Introduction

High Performance Computing (HPC) applications are usually *Single Process-Multiple Data* (SPMD) and are implemented using an MPI or an OpenMP library. In MPI applications, all the processes execute the same code on different data sets and use synchronization primitives (such as barriers or collective operations) to coordinate their work. Since the processes execute the same code, they are supposed to reach their synchronization points roughly at the same time. However, this is not always the case. Some applications among those running on MareNostrum, the 13th supercomputer on the Top500 list, installed at the Barcelona Supercomputing Center (BSC), suffer from *imbalancing*, i.e. the execution time of the processes in the parallel application is not the same (in Section II we will see some causes of applications' imbalancing). Therefore, if a process runs for longer than the others belonging to the same application, all the other processes have to wait for that process to complete its execution. During this time the CPUs of the waiting processes are idle, thus, not performing any useful job. As an example, let us assume that one process has to complete its execution while all the other processes are waiting for it to reach the synchronization point; then, in MareNostrum, up to 10239 processor may be idle, resulting in a significant loss of performance and waste of resources.

The performance achievable by traditional superscalar processor designs has almost saturated due to the limitation imposed by Instruction-Level Parallelism (ILP). As a consequence, Thread-Level Parallelism (TLP) has become a common strategy for improving processor performance. Since it is difficult to extract more Instruction-Level Parallelism from a single program, Multi-Threaded (MT) processors obtain more parallelism by simultaneously executing several tasks. This strategy has led to a wide range of MT processor architectures, from Simultaneous Multi-Threaded processors (SMT) [22], [28], [33], in which most processor resources are shared among threads, to Chip Multi-Processors [5] (CMP), in which every thread has its own dedicated processor resources, only sharing the highest levels of the memory hierarchy (for example the L2 cache), and a combination of both [30]. Resource sharing makes multi-threaded processors have good performance/cost and performance/power consumption ratios [3], two desirable characteristics for a Supercomputer. As a consequence, most of the current Supercomputers already use processors with some multi-threaded features [1].

Usually, software has no control over how processor resources are distributed among running threads in multi-threaded processors. For example, in an SMT processor the *instruction fetch policy*, decides how instructions are fetched from the threads, thereby implicitly determining the way internal processor resources are allocated to the threads. This is an undesirable characteristic that makes the execution time of programs unpredictable [6]. In

[1]In this paper the term MT processors refers to both multi-core (CMP) and multi-thread (SMT) processors.

order to alleviate this problem, recently, some processor vendors have equipped their MT processors with mechanisms that allow the software to control processor's internals resource allocation, and thus, control application's speed. Our view is that these mechanisms open new opportunities to improve applications performance as they offer fine-grain ways to control the progress of applications by allocating or deallocating processor resources to them.

This paper is a first step towards that direction. We show how re-assigning hardware resources in a multithreaded processor can reduce the imbalance in parallel applications, and hence improving performance. In particular we propose a way to regain balance assigning more hardware resources to processes that computes for more time, reducing their execution time and, thus, the waiting time of all the other processes belonging to the same HPC application. This solution is transparent to the users: since the solution is at Operating System (OS)/hardware level, users do not need to know the processor's implementation details at compile time nor to adapt their programming model in order to use our proposed solution. To the best of our knowledge, this is the first time that such a solution is implemented in a real machine.

We explored this idea experimentally on a real system with a MT processor, the IBM POWER5 [30]. The POWER5 is a dual-core, 2-way SMT processor that allows us to change the way hardware resources are assigned to the core's contexts by means of a *thread context priority* (or hardware thread priority[2]) that controls the number of resources each context receives. This machine runs a Linux kernel that we had to modify in order to allow the HPC application to exploit the advantage of re-assigning the processor's resources. We performed several experiments with MPI applications:

1) We started from a micro-benchmark (MetBench), developed at BSC, where we introduce some artificial imbalancing.
2) In the second experiment, we ran the widely used the BT-MZ NAS [24] benchmark; this version suffers of imbalance.
3) Finally, we used a real application running on MareNostrum, SIESTA [29].

Our results show an improvement of 18% for the NAS benchmark and 8.1% for SIESTA. In addition, our results also show that this mechanism of controlling hardware resources is a powerful tool that, if used incorrectly, may lead to significant performance loss. Moreover, non-HPC applications may benefit differently from re-assigning hardware resources or not at all.

[2]The hardware thread priorities mentioned here are independent of the operating systems concept of thread priority.

The rest of this paper is organized as follows: Section II shows the imbalancing problem in HPC applications; Section III presents similar works in the same area; Section IV introduces our solution based on smart allocation of hardware resources; Section V introduces the architecture details of the IBM POWER5 processor that allow us to implement and try our proposal; Section VI describes the changes we applied to the Linux kernel; Section VII shows our set of experiments on the IBM POWER5 system for our micro-benchmark, a standard benchmark and a real application; finally Section VIII provides our conclusion and future work.

II. Imbalance in HPC applications

HPC applications are usually SPMD, which means that every process executes the same code on different data. For example, let us assume that an HPC application is performing a matrix-vector multiplication and that each process receives a sub-matrix and the part of the vector required to compute the sub-matrix by vector multiplication. If the matrix can be divided into homogeneous parts (i.e., they require the same amount of time to be processed), all the processes in the parallel application would finish, ideally, at the same time.

However, the data set could be very different: let us say that, in the previous example, the matrix is sparse or triagonal, hence, the time required to process the data sub-set could vary as well. In this scenario the amount of time required to complete the sub-matrix by vector multiplication depends on the number of non-zero values present in the sub-matrix. In the extreme case, one process could receive a full sub-matrix while another an empty one. It is clear that the former process requires much more time to reach the synchronization point than the latter.

We classify the sources of imbalancing in two main classes: *intrinsic* and *extrinsic* factors of imbalancing.

A. Intrinsic imbalancing

We refer to *intrinsic imbalancing* as the imbalancing an application experiences because of data (for example a sparse matrix) or algorithm (master-slave architecture) imbalancing. The causes for the imbalancing are internal to the application's code, input set or both.

The intrinsic imbalancing could be caused by several factors, here we point some of them out:

Input set: As we already said, this scenario happens when a process has a small input set to work on while another has a large amount of data to process.

Domain: Iterative methods approximate the solution of a problem (for example, Partial Differential Equations, PDE) with a function in some domain starting from

an initial condition. The domain is divided in several sub-domains and each process computes its part of the solution. At the end of every iteration, the error made in the approximation is computed and, eventually, another iteration is to be started. If the function in some part of the domain is smooth, only few iterations are required to converge to a good approximation. Conversely, if the function has several picks in the sub-domain, more iterations are necessary to find a good solution and/or more points in the domain have to be considered during the computing phase.

Data exchanging: During their execution, processes may require to exchange data among themselves. If the two peers are on the same node, the latency of the communication is small; if a process needs to exchange data with a neighbor on another node the latency is large, even larger if the destination process is far away in the network.

In all the previous cases, and the other ones that we do not mention for lack of space, the application might result to be imbalanced.

B. Extrinsic imbalancing

Even if both the application's algorithm and the input set are balanced, the execution of the parallel application could still be imbalanced. This is caused by external factors that slow some processes down (but not others). For example, the Operating System (OS) might decide to run another process (say a kernel daemon) in place of the MPI process running on one CPU. Since that MPI process is not able to run all the time while the others are running, it takes more time to complete, making all the other processes waiting for it.

Those external factors are the sources of *extrinsic imbalancing*. Even in this case, there might be several causes for the imbalancing:

OS noise: The CPU is used by the OS to perform services such as handling interrupts, reclaiming memory, assigning memory on demand, etc. The OS noise has been recognized as one of the major source of extrinsic imbalancing [11], [25], [32]. A classical example is the interrupt annoyance problem present in Intel processors: all the interrupts coming from external devices are routed to CPU0, therefore, the OS noise caused by executing the interrupt handlers on CPU0 is higher than the noise on the other CPUs.

User daemons: It is common that HPC systems also run profile or statistic collectors together with the HPC application. These activities could steal computing power from one process, actually, delaying it.

Network topology: Exchanging data between processes in the same sub-network is faster than exchanging data between processes in different sub-network; the same rule applies to processes communicating inside a NUMA domain versus processes running in different NUMA domains. In general, if the job scheduler has placed processes that need to communicate "far away", their communication latency could increase so much that the whole application will be affected.

An expert programmer could reduce the intrinsic imbalance in the application. However, this is not an easy task, for the imbalance could be caused by the input data set, not only by the algorithm, and, therefore, it could only appear with some input and be absent with others. Even worse is the case of extrinsic imbalance: those factors are neither under the control of the application nor of the programmer and there is no straightforward way to solve this problem. Thus, it is clear that a mechanism that aims to solve the imbalance of an application should be transparent to the user, regardless of the programming model, libraries or input set.

III. Related work

Traditional solutions to attack the problem of load imbalance in HPC applications typically use dynamic data re-distribution. For OpenMP applications load balancing may be performed using some of the existing loop scheduling algorithms that assigns iterations to threads dynamically [4]. MPI applications are much more complex because data communications are defined explicitly in the algorithm by programmers. Static approaches for distributing data using sophisticated tools have been proposed: for example, METIS [2] analyzes data and tries to find the best data distribution. These approach achieve good performance results but have the drawback that they must be repeated for each input data set and architecture. Dynamic approaches have also been proposed in the literature (Schloegel et al. [27] and Walshaw et al. [34]). The authors try to solve the load-balancing problem of irregular applications by proposing mesh repartitioning algorithms and evaluating the convenience of repartitioning the mesh or adjust it.

Resource re-distribution is another approach that consists of assigning more resources to those processes that compute for more time. In the case of OpenMP, this can be useful when using nested parallelism, assigning more threads to those groups with high load [9]. The case of MPI is much more complex because 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). This problem has been also approached through hybrid programming models, combining MPI and OpenMP. Huang and Tafti [12] balance irregular applications by modifying the computational power rather than using the typical mesh redistribution. In their work, the application detects the

overloading of some of its processes and tries to solve the problem by creating new threads at run time. They observe that one of the difficulties of this method is that they do not control the operating system decisions which could oppose their own ones.

Concerning the use of SMT architectures for HPC applications, several studies [8], [7] show that Hyper-Threading (the SMT implementation of Intel Processors) improve performance for some workloads. However, for other workloads there are many conflicts when accessing shared resources, creating a negative impact on the performance. In [8] the study is performed for MPI applications while in [7] the study focuses in OpenMP applications. In [7] the authors propose a mechanism that, given a multiprocessor machine with Hyper-Threading processors, dynamically deactivates the Hyper Threading in some processors in order to improve the performance of the workload under study.

Our proposal is orthogonal to both the thread redistribution and the dynamically activating Hyper-Threading. Let us assume that we want to run an HPC application on a cluster having several IBM POWER5 processors. The proposal in [7] can be used to determine in which cores SMT has to be deactivated. For those cores with the SMT feature active, our proposal can be used to select the appropriate hardware priority to reduce imbalancing. Compared with thread-distribution, our contribution can be seen as low level solution for load balancing.

IV. Our Proposal

Balancing a HPC application by hand is a time-consuming task and may require quite a lot of effort. In fact, the programmer has to distribute the data among the processes considering the way the algorithm has been implemented and the correctness of the application. Moreover, this work has to be done for each application and, likely, every time the input changes. As we will see later, our proposal is transparent to the user and independent from the applications or the input set.

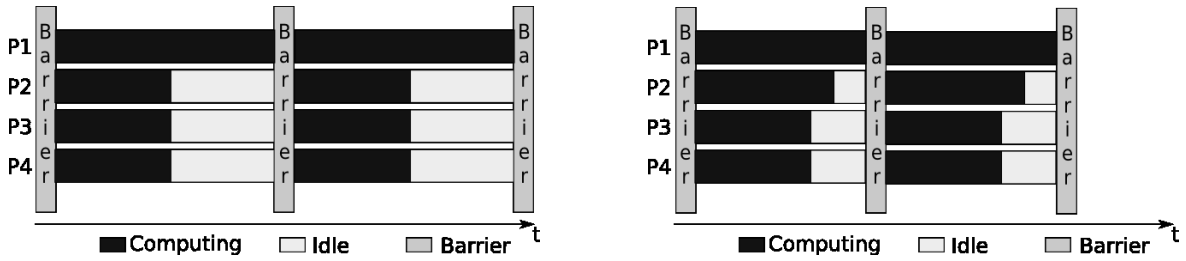
With the arrival of MT architectures, and in particular those that allow the software to control processor's resource allocation, new opportunities arise to mitigate the problem of imbalancing in HPC systems. This is mainly due to the fact that the software is allowed to exercise a fine control over the progress of tasks, by allocating or deallocating processor resources to them. Such a transparent, fine-grain control cannot be achieved by previous solutions for load imbalancing; in fact, even if a lot of processors with shared resources have been introduced in the market since early 90s, very few of them allow the software to control how internal resources are shared. We think that allowing the software to control how to assign shared resources is a key factor for HPC systems. In this view, having MT processor able to provide such mechanism will be essential for improving the performance of HPC systems.

Our solution for balancing HPC applications consists of assigning more hardware resources to the most compute-intensive processes (the bottleneck). Giving this process more hardware resource shall decrease its execution time and, since this process is the bottleneck of the application, the execution time of the whole MPI application.

Clearly the underlying processor has to support this capability to re-assign processor resources among running threads. Currently, multi-threaded processors like the IBM POWER5 [30] and POWER6 [21] or the Cell processor [13], [14] provide such a capability with their thread priority mechanisms: the higher the priority of one context, the higher the amount of resources it receives. In this paper, we focus on the IBM POWER5 but our idea is general and can be also applied to other MT processors that allow the OS to the allocation of processor's resources (for example, partitioning a shared L2 cache in a multi-core CPU [23], [26]). The IBM POWER5 processor is used, among others, by ASC Purple, the 11th supercomputer in the Top500 list installed at the Lawrence Livermore National Laboratory.

We should point out that increasing the performance of one process by giving it more hardware resources, does not come for free. In fact, at the same time, the performance of the other process running on the same core, therefore sharing the resources with the former process, reduces. Figure 1 shows a synthetic example that illustrates this case: in Figure 1(a) $P1$ shares resources with $P2$, while $P3$ shares them with $P4$; $P2$, $P3$ and $P4$ take the same amount of time to reach their synchronization point but $P1$ takes much more time. As a result $P2$, $P3$ and $P4$ are idle for a long time. In Figure 1(b) $P1$ uses more hardware resources and its execution time decreases; $P2$'s execution time, instead, increases since it runs with less hardware resources. Still $P2$ has enough "spare time" and its slowdown does not affect the application's performance because it is not the bottleneck. On the other hand, the performance improvement of $P1$ directly translates into a performance improvement for the whole application, as it is possible to see confronting Figures 1(a) and 1(b).

Finally we would like to point out that we made no assumption on what kind of application, the programming model or the input set the programmer has to use. Our only assumption regards the underlying processor, which must provide a shared resource control mechanism. Besides that, our solution is at OS level and completely transparent to the users, who are free to use



(a) Imbalanced HPC application

(b) More balanced HPC application

Fig. 1. Expected effect of the proposed solution

the MPI, OpenMP or whatever programming model or library they wish. Moreover the approach is dynamic and the amount of resources assigned to a process can change according to the input set provided to the application.

V. The IBM POWER5 processor

The IBM POWER5 [15], [16], [17] processor is a dual-core chip where each core is a 2-way multi-thread [19] core. Each core has its own private first-level data and instruction cache. The unified second- and third-level caches are shared between cores.

The forms of Multi-Threading implemented in the POWER5 are Simultaneous Multi-threading and Chip-Multiprocessing. The main characteristic of SMT processors is their ability to issue instructions from the different threads in the same cycle. As a result, SMTs not only can switch to a different thread to use the idle issue cycles in a long-latency operation, like coarse-grain multi-threading, or in a short-latency operation, like in a fine-grain multi-threaded, but also fill unused issue slots in a given cycle.

What makes the IBM POWER5 ideal for testing our proposal is the capability that each core has to assign some hardware resources to one context rather than to the other. Each context in a core has a *hardware thread priority*, 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), as illustrated in Table I. As the hardware thread priority of a context increases (keeping the other constant) the amount of hardware resources assigned to that context increases too.

A. Thread priorities implementation

The way the core processor assigns resources to each thread is by decoding more instructions from one context than from the other. The number of decode cycles assigned to each thread 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).

TABLE I
Hardware thread priorities in the IBM POWER5 processor

Priority	Priority level	Privilege level	or-nop inst.
0	Thread shut 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`

Let us assume two threads (ThreadA and ThreadB) are running on a POWER5 core with priorities X and Y, respectively. In POWER5 the decode time is divided in time-slices of R cycles: he lower priority thread receives 1 of those cycles, while the higher priority thread receives $(R-1)$ cycles. R is computed as:

$$R = 2^{|X-Y|+1}$$

Table II shows the possible values of R and how many decode slots are assigned to the two threads as the difference between ThreadA's and ThreadB's priority moves from 0 to 4. In fact, the amount of resources assigned to a thread is determined using the difference between the thread priorities, X and Y. For example, assuming that ThreadA has hardware priority 6 and ThreadB has hardware priority 2 (the difference is 4), then the core fetches 31 times from context0 and once from context1 (more details on the hardware implementation are provided in [10]). It is clear that the performance of the process running on context0 shall increase to the detriment of the one running on context1.

When any of the threads has priority 0 or 1, the behavior of the hardware prioritization mechanism is different, as shown in Table III.

B. Hardware interface for priority management

In POWER5 the hardware priority is assigned to threads by software and can be changed at run-time. A thread priority can range from 0 to 7, where 0 means the

TABLE II
DECODE CYCLES ALLOCATION IN THE IBM POWER5 WITH DIFFERENT PRIORITIES

Priority difference (X-Y)	R	Decode cycles for A	Decode cycles for B
0	2	1	1
1	4	3	1
2	8	7	1
3	16	15	1
4	32	31	1

TABLE III
RESOURCE ALLOCATION IN THE IBM POWER5 WHEN THE PRIORITY OF ANY OF THE THREADS IS 0 OR 1

Thr.A	Thr.B	Action
> 1	> 1	Decode cycles are given to the two threads as according with the thread's priorities
1	> 1	ThreadB gets all the execution resources; ThreadA takes what is left over
1	1	Power save mode; both ThreadA and ThreadB receive 1 of 64 decode cycles
0	> 1	Processor in ST mode. ThreadB receives all the resources.
0	1	1 of 32 cycles are given to ThreadB
0	0	Processor is stopped

thread is switched off and 7 means the thread is running in ST mode (i.e., the other thread is off). The supervisor (OS) can set 6 of the 8 mentioned priorities, from 1 to 6; user software can only set priority 2, 3, 4; the Hypervisor can always span the whole range of priorities.

The IBM POWER5 provides two different interfaces to change the priority of a thread: issuing an `or`-nop instruction or using the *Thread Status Register* (`TSR`). We used the former interface, in which case a thread has to execute an instruction like `or X,X,X`, where `X` is a specific register number (see Table I). This operation does not do anything but changing the hardware thread priority. Besides the priorities, Table I shows the privilege level required to set each priority and how to change priority using this interface. The second interface consists of writing the hardware priority into the local (i.e., the thread) `TSR` by means of an `mtspr` operation. The actual thread priority can be read from the local `TSR` using a `mfspr` instruction.

VI. THE LINUX KERNEL INTERFACE

Users can only set three priorities: MEDIUM (4), MEDIUM-LOW (3) and LOW (2). This basically means that users are only allowed to reduce their priority, since the MEDIUM priority is the default case. If the user reduces the thread priority when a process does not require lot or resources (for example because the process is waiting for a lock), the overall performance might increase (because the other thread will receive more resources and, therefore, might go faster). Thus, it is recommended that the user reduces the thread priority whenever the thread processor is executing a low-priority operation (such as spinning for a lock, polling, etc.).

Modern Linux kernels running on IBM POWER5 processors make use of the hardware priority mechanism the chip provides. In this Section we will first explore the standard behavior of the Linux kernel when dealing with hardware priorities, and then present how we modified the standard kernel in order to solve our imbalancing problem by means of the IBM POWER5 hardware prioritization mechanism.

A. Using priorities in the standard Linux kernel

The Linux kernel only exploits hardware priorities in a limited number of cases: the general idea is to reduce the priority of a process that is not performing any useful operation and to give more resources to the process running on the other context.

The standard Linux kernel makes use of the thread priorities in three cases:

1) The processor is spinning for a lock in kernel mode. In this case the priority of the spinning process is reduced (the process is not really advancing in its job).
2) The kernel is waiting for some operations to complete. This happens, for example, when the kernel wants a specific CPU to perform some operation by means of a `smp_call_function()` (for example, invalidating its TLB) and cannot proceed until the operation has completed. In this case the priority of the CPU is decreased until the operation completes.
3) The kernel is running the idle process because there is no other process ready to run. In this case the kernel reduces the priority of the idle CPU and, eventually, put the core in Single Thread (ST) mode.

In all these cases the kernel reduces the priority of the context, restoring the priority to MEDIUM when there is some job to perform. The hardware thread priority is also reset to MEDIUM as soon as the kernel executes an interrupt or an exception handler as well as a system call. In fact, since the kernel does not keep track of the actual priority, it cannot restore the current priority. Therefore, the kernel simply resets the priority to MEDIUM every time it starts to execute an interrupt handler (or a system call), so that it can be sure that those critical operations will be performed with enough resources.

B. Modification on the Linux kernel

In order to check that our approach can be used for balancing the HPC application, we had to modify the original kernel code for two reasons:

1) every time the CPU receives an interrupt, the interrupt handler sets the priority back to MEDIUM, regardless of the current priority. We want to maintain the given priority even after an interrupt is received or during the interrupt handler itself; thus, we removed the code that makes use of the hardware thread priority capabilities from the handlers.
2) only hardware priorities 2 (LOW), 3 (MEDIUM-LOW) and 4 (MEDIUM) can be set by a user-level program. Priorities 1 (VERY LOW), 5 (MEDIUM-HIGH) and 6 (HIGH) can only be set by the Operating System (OS). Priorities 0 (context off) and 7 (VERY HIGH, ST mode) can only be set by the Hypervisor. We developed an interface that allows the user to set all the possible priorities available in kernel mode. A user who wants to set priority `N` to process `<PID>` shall simply write to a `proc` file, like:

   ```
   echo N > /proc/<PID>/hmt_priority
   ```

In this moment the patch only provides a mechanism to set all the priorities (available at OS level) from user applications. It is responsibility of the user applications (or run time systems) to balance the system load using this interface. This is the first step to prove that our proposal is a good solution for the problem of imbalancing in HPC. Our next step will be to have a system which dynamically changes the priority of the running processes so that more resources are assigned to the most intensive processes automatically. We developed the patch for the standard Linux 2.6.19.2 so that it is not intrusive and has no impact on the performance of our experiments.

VII. Experimental Results

In order to validate our proposal we performed experiments on an IBM OpenPower 710 server, which has one POWER5 processor.

Since MPI is the most common protocol, we tested our proposal using MPI applications (in the experiments we used the MPI-CH 1.0.4p1 implementation of MPI protocol).

We present our results for three different cases: Section VII-A shows how the IBM POWER5 priority mechanism works using our micro-benchmark (MetBench); Section VII-B provides details on how we used the hardware priorities to balance a widely used benchmark (BT-MZ) and improve its performance. Finally Section VII-C presents the results for a real application frequently executed on MareNostrum (SIESTA).

In order to present experiments in a simple way, we used two metrics: first, the percentage of imbalance (computed as the maximum waiting time in percentage of the processes in the MPI application); second, the total execution time of the 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.

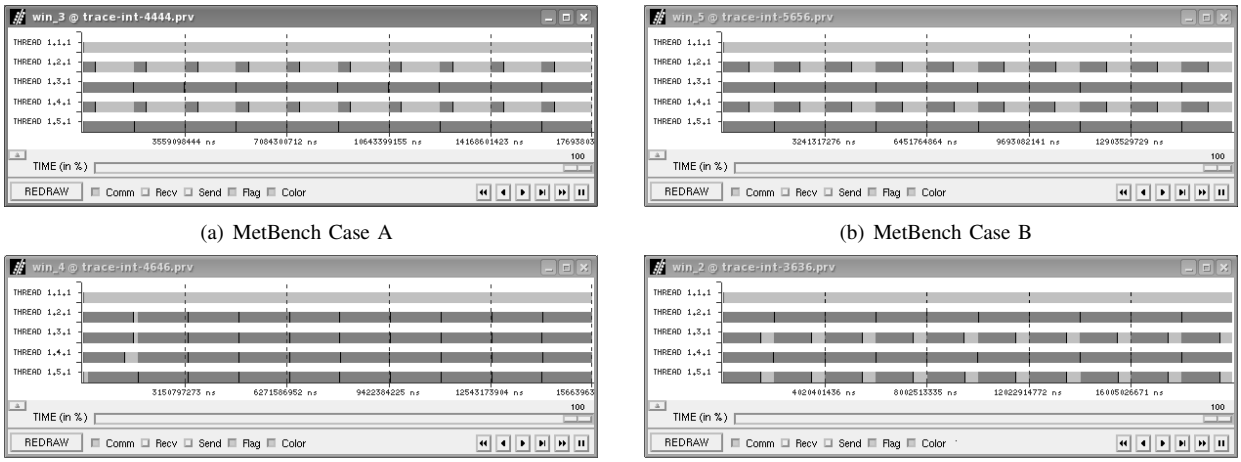
A. MetBench

MetBench (Minimum Execution Time Benchmark) is a suite of MPI micro-benchmarks developed at BSC whose 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 wait for all the others to complete their task. The role of the master is to maintain a strict synchronization between the workers: once all the workers have finished their tasks, the master eventually starts another iteration (the number of iterations to perform is a run time parameter). The master and the workers only exchange data during the initialization phase and use an `mpi_barrier()` to get synchronized.

One of the goals of MetBench is to allow researchers at BSC to understand the performance and capabilities of a processor or a cluster. In order to do that, we developed several loads, each one stressing a different processor resource (the Floating Point Unit, the L2 cache, the branch predictor, etc) for a given amount of time.

In this experiment we introduce imbalancing in the MPI application by assigning to a worker a larger load than the one assigned to the worker 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. As we will see in Section VII-B and Section VII-C this scenario is quite common for both standard benchmarks and real applications. Figure 2 shows the effect of the proposed solution on MetBench. Each horizontal line represents the activity of a process and each color represents a different state: dark-grey bars show computing time while light-grey bars show waiting time (at the end of each computation phase there is a black bar that represents statistical operations). In this example, processes $P1$ (the master), $P2$, and $P3$ are mapped to the first core of the POWER5, while processes $P4$ and $P5$ are mapped to the other core. The x-axis represents time.

Case A: Figure 2(a) represents our reference case, i.e., the MPI application is running with default priorities (4). As we can see from figure 2(a) MetBench shows a great imbalance (75.69%, as reported in table IV): more specifically, processes $P1$ and $P3$ spend most of their time waiting for processes $P2$ and $P4$ to complete their computing phase.



(a) MetBench Case A

(b) MetBench Case B

(c) MetBench Case C

(d) MetBench Case D

Fig. 2. Effect of the proposed solution on MetBench. Each trace represents only some iterations of the application.

Case B: Using our solution we increased the priority of $P2$ and $P4$ (the most computing intensive processes) up to 6, while the priority of $P1$ and $P3$ are set to 5 (remember from Section V-A that what really matters is the difference between the thread priorities, here $P1$ and $P3$ are running with less priority than in Case A). Figure 2(b) shows how the imbalance has been reduced from 75.69% to 48.82%; this improvement translates to a shorter total execution time (from 81.64 sec to 76.98 sec, 5.71% of improvement).

Case C: Then we tried to reduce again the amount of hardware resources assigned to $P1$ and $P3$, hoping to speed $P2$ and $P4$ up. Indeed, we obtained an even more balanced situation where all the processes compute for (roughly) the same amount of time (the imbalance is 1.96%). The total execution time reduces to 74.90 sec (8.26% of improvement over Case A).

Case D: Next, we reduced again the amount of resources given to $P1$ and $P3$. As we can see from Figure 2(d) we reversed the imbalance, i.e., now $P2$ and $P4$ are so much faster than $P1$ and $P3$ that they spend most of their time waiting. As a result both the imbalance (26.62%) and the execution time (95.71 sec) increase.

Case D shows an interesting properties of the IBM POWER5 hardware priority mechanism: the hardware thread priority implementation is a powerful tool but the performance of the penalized process can be reduced much more than linearly (in fact, exponentially), thus, it could become the new bottleneck.

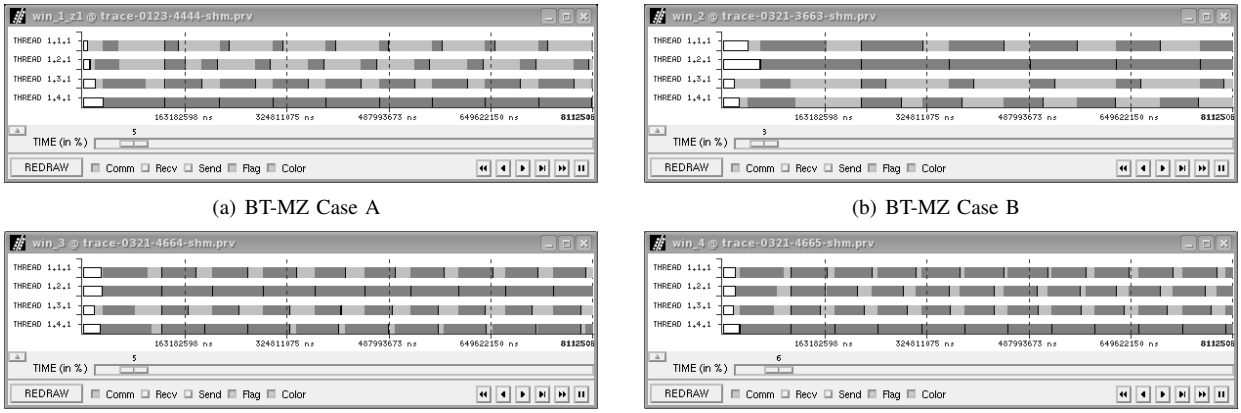
TABLE IV
METBENCH BALANCED AND IMBALANCED CHARACTERIZATION

Test	Proc	Core	P	Comp %	Sync %	Imb %	Exec. Time
A	P1	1	4	24.32	75.67	75.69	81.64s
	P2	1	4	98.99	1.00		
	P3	2	4	24.31	75.69		
	P4	2	4	99.99	0.00		
B	P1	1	5	51.16	48.83	48.82	76.98s
	P2	1	6	99.82	0.18		
	P3	2	5	51.18	48.81		
	P4	2	6	99.98	0.01		
C	P1	1	4	98.96	1.03	1.96	74.90s
	P2	1	6	98.56	1.43		
	P3	2	4	97.01	2.99		
	P4	2	6	98.37	1.63		
D	P1	1	3	99.87	0.12	26.62	95.71s
	P2	1	6	73.25	26.74		
	P3	2	3	99.72	0.27		
	P4	2	6	73.25	26.74		

B. 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 our MetBench benchmark: 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()` and `mpi_irecv()`); after this communication phase (which lasts for a very short time, around 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



(a) BT-MZ Case A

(b) BT-MZ Case B

(c) BT-MZ Case C

(d) BT-MZ Case D

Fig. 3. Effect of the proposed solution on BT-MZ. Each trace represents only some iterations of the application.

neighbors (note that this does not mean that each process gets synchronized with all the other processes). Once a process has exchanged all the data it had to exchange, a new iteration can start and the previous behavior repeats again till the end of the application (in our experiments we used BT-MZ with default values: class A with 200 iterations).

Case A: Figure 3(a) shows the BT behavior in the reference case, i.e. when process P_i is assigned to CPU_i and the priority of all the processes is 4. After an initialization phase (white bars at the beginning of the execution of each thread), all the processes reach a barrier (synchronization point). From this point on, the real algorithm starts: during every iteration, each process alternate computing phases (dark grey) with synchronization phases (light grey) at the end of communication phases (black).

It is easy to see from figure 3(a) that BT-MZ shows a great imbalance (82.23%), as it is also reported in table V[3].

The imbalance is caused by the fact that some processes (for example process $P1$) have a small part of the data to work on, while other processes (for example, processes $P4$) have a large amount of data to take care of. It is also clear that process $P4$ is the bottleneck of the application and that speeding up this process will improve overall performance.

In order to solve the imbalance introduced by data repartition in BT-MZ, we ran process $P1$ and $P4$ on the same core and assigned more hardware resources to the latter, improving its performance while decreasing $P1$'s performance. This mapping should allow us to give a large amount of resources to process $P4$ without reversing the imbalance, i.e., without making process $P1$ slower than $P4$ like it was the case for MetBench (Case D). In fact, this mapping seems reasonable, for our goal is to increase the performance of $P4$ (the most computing intensive process) and we know that, with this operation, we will reduce the performance of the process running on the same core with $P4$. We chose $P1$ because it is the process with the shortest computation phase.

[3]Even if the goal of this paper is not to show whether SMT processors are useful in HPC or not, the table also shows the ST mode performance (only one process per core) of the application.

TABLE V
BT-MZ BALANCED AND IMBALANCED CHARACTERIZATION

Test	Proc	Core	P	Comp %	Sync %	Imb %	Exec. Time
ST	P1	1	7	49.33	50.59	50.27	108.32s
	P2	2	7	99.46	0.32		
A	P1	1	4	17.63	82.32	82.23	81.64s
	P2	1	4	28.91	71.02		
	P3	2	4	66.47	33.4		
	P4	2	4	99.72	0.09		
B	P1	1	3	52.33	47.49	70.93	127.91s
	P2	2	3	99.64	0.14		
	P3	2	6	28.87	71.07		
	P4	1	6	46.26	53.65		
C	P1	1	4	65.32	34.48	45.99	75.62s
	P2	2	4	99.68	0.12		
	P3	2	6	53.78	46.11		
	P4	1	6	85.88	14.44		
D	P1	1	4	82.73	17.10	33.38	66.88s
	P2	2	4	73.68	26.17		
	P3	2	5	66.40	33.47		
	P4	1	6	99.72	0.09		

Case B: In our first attempt to reduce the imbalance we assigned priority 3 to processes $P1$ and $P2$ and priority 6 to processes $P3$ and $P4$. Figure 3(b) shows how 1) the imbalance has been inverted (process $P1$ now takes longer than $P4$ and 2) the new bottleneck is now process $P2$, which is also running with priority 3. Even if the imbalance has been reduced (from 82.23% to 70.93%), the total execution time now takes longer (127.91 sec instead of 81.62 sec), which means the new

bottleneck runs for much more than the previous one.

Case C: In order to restore the original relative behavior between process $P1$ and $P4$ we incremented the resources assigned to process $P1$. Figure 3(c) shows that $P1$ now runs for less time than $P4$, as in Case A. The imbalanced has been reduced from 70.93% of Case B to 45.99% in Case C and giving more resource to $P2$ (which is again the bottleneck) reduced the total execution time to 75.62 sec, with a 7.37% of improvement with respect to Case A.

Case D: Finally, we can argue that $P2$ and $P3$ execute their operation on a similar amount of data, therefore the amount of resources given to each process should not be as different as for $P1$ and $P4$. In our last test, we still assigned priority 4 to $P1$ and 6 to $P4$, as in the previous case, but we assigned priority 5 to $P2$ and 6 to $P3$, sharing resources between these two processes running on the same core more equally. Figure 3(d) shows that the imbalance has been reduced again with respect to Case C, in fact, now $P2$ and $P3$ compute more or less for the same amount of time. Also the new bottleneck is $P4$, which takes much shorter than $P2$ in Case C. Table V shows how the total execution time has also been reduced to 66.88 sec, with a 18.08% of improvement over the reference Case A.

C. Siesta

Our last experiment consists of running SIESTA as an example of real application. SIESTA [31] 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.

The application presents an imbalance caused by both the algorithm and the input set. SIESTA behavior, however, is not constant during each iteration, as can be seen in Figure 4(a); this makes our static balancing solution not as good as for the BT-MZ case. Yet, we achieved a improvement of 8.1% with respect to the reference case (Case A).

Case A: Like for BT-MZ, Case A is the reference case, i.e., where process P_i is assigned to CPU_i and the priority of all the processes is set to 4. Figure 4(a) shows the trace for this reference case. The program starts with an initialization phase (11.99% of the total time) at the end of which each process in the application must reach a barrier. The initialization phase already presents some little imbalance, which evidences how the input set makes SIESTA not balanced. In the internal parts, each process exchanges data only with a subset of the other processes in the application, and then reaches a synchronization point (`WaitAll()`), waiting for all the others to complete their jobs. In the last part, the processes finalize their work (13.41% of the total time): after the last barrier, each process computes its function on its sub-set of data and then ends. A complete execution of the program in this configuration takes 858.57 secs.

TABLE VI
SIESTA BALANCED AND IMBALANCED CHARACTERIZATION

Test	Proc	Core	P	Comp %	Sync %	Imb %	Exec. Time
ST	P1	1	7	81.79	14.22	8.88	1236.05s
	P2	2	7	93.72	5.34		
A	P1	1	4	75.94	15.42	14.43	858.57s
	P2	1	4	75.24	18.11		
	P3	2	4	82.08	10.71		
	P4	2	4	93.47	3.18		
B	P1	2	4	79.57	14.67	5.99	847.91s
	P2	1	4	87.06	10.15		
	P3	1	5	72.04	12.69		
	P4	2	5	77.73	8.68		
C	P1	2	4	83.04	10.59	1.46	789.20s
	P2	1	4	79.66	10.52		
	P3	1	4	80.78	9.41		
	P4	2	5	78.74	9.13		
D	P1	2	4	90.76	5.60	16.64	976.35s
	P2	1	4	65.74	22.25		
	P3	1	4	68.08	19.36		
	P4	2	6	63.95	18.10		

Case B: As we can see from the trace in Figure 4(a) is not easy to understand how to balance the application and whether our balancing approach is worth. However, Table VI shows some more information about SIESTA (hard to retrieve from the trace): processes $P1$ and $P2$ spend a considerable amount of time waiting for $P3$ and $P4$ to reach the barrier. Thus, the first hint would be to put $P1$ and $P3$ on one core and $P2$ and $P4$ on the other and then play with priority. We tried this case but then we realized that $P2$ and $P3$ have almost the same amount of data to work on. Thus, in Case B we put $P2$ and $P3$ on the first core and $P1$ and $P4$ on the second one and increased the priority of $P3$ and $P4$ to 5. In this case we achieved a little improvement of 1.24% (the total execution time is 847.91 sec). Figure 4(b) shows that, in this new configuration, $P2$ is the new bottleneck of the finalization part.

Case C: In the previous case we obtained a little improvement, still the application results quite imbalanced. We realized that, since $P2$ and $P3$ work, more or less, on the same amount of data, using a different priority for these two processes may introduce even more imbalance. Figure 4(b) shows that, indeed, this is the case. In Case C we restored the original relative behavior between process $P2$ and $P3$ setting both their priority to 4 (i.e., the difference is 0). Figure 4(c) shows how the application is now more balanced. For

(a) SIESTA Case A

(b) SIESTA Case B

(c) SIESTA Case C

(d) SIESTA Case D

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

example, looking at the initialization and the finalization part, it is possible to see that the processes are much more balanced than in Case A and Case B. In fact, re-balancing SIESTA reduces the total execution time to 798.20 sec, an improvement of 8.1% with respect to the reference case.

Case D: Following the same idea of the previous case (i.e., leave $P2$ and $P3$ with the same priority and play with $P1$ and $P4$), we increased the amount of resources assigned to $P4$, penalizing $P1$. Figure 4(d) shows how we reverse the imbalance: SIESTA is again imbalanced, though in a different way than in the reference case. In Case D, $P1$ (the process with less hardware resources) is the bottleneck (in the initialization, finalization and most of the internal phases) and the total execution time increases to 976.35 sec, with a loss of 13.72%.

BT-MZ and SIESTA are two cases of non-balanced HPC applications, though their imbalance is quite different. BT-MZ executes several iterations, all of them similar from the execution time, CPU utilization and imbalance point of view. SIESTA also executes several iterations but each iteration is not necessarily similar to the previous or the next one. In particular, the process that computes the most is not the same across all the iterations. For example, in the *i*-th iteration $P1$ could be the bottleneck while in the *(i+1)*-th the most computing process could be $P4$. This behavior suggests that a good balancing mechanism would prioritize $P1$ in the *i*-th and $P4$ in the *i+1*-th iteration. Our static approach does not allow us to play in this way as we assign the priority at the beginning of the execution and never change them during the execution. We argue that a dynamic mechanism is required to correctly set priorities for applications that change their behavior throughout their execution. Since real applications are likely to behave like SIESTA rather than like BT-MZ, we intend to extend our balancing mechanism as part of the Operating System, so that the OS can dynamically set the priority of each process according to actual application behavior.

VIII. Conclusions and Future Work

In this paper we showed how allowing software to control the amount of shared resources assigned to each thread in a MT processor may improve the performance of HPC applications. In fact, some applications show an imbalanced behavior, i.e., some processes require more time to complete their computing phase while all the other processes are waiting at some synchronization point and cannot move forward. While the imbalancing can be caused by either external or internal factors (most likely both), it is clear that it may reduce the performance of an HPC application, resulting in a significant waste of resources in Supercomputers. Our results show how using our modified Linux kernel to control a processor capable to dynamically assign processor resources to running threads (the IBM POWER5 in our case), reduces the application imbalancing and, therefore, improves overall performance. The experiments we performed show an improvement up to 18% for a widely used BT-MZ benchmark and up to 8.1% for a real application. We achieved these results without putting the burden of balancing the application on the programmer and regardless of the used programming model.

Our results suggest that an automatic mechanism could even increase the actual improvement, thus, motivating the use of MT processors with the capability to re-assign hardware resources between threads in future Supercomputers. We plan to extend our OS by introducing an algorithm that will automatically detect if a process deserves an higher amount of resources and which process should be deprived of those resources so that imbalancing can be reduced.

ACKNOWLEDGMENT

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN-2004-07739-C02-01, TIN-2007-60625, the HiPEAC European Network of Excellence and a Collaboration Agreement between IBM and BSC with funds from IBM Research and IBM Deep Computing organizations. Carlos Boneti is granted by the Catalonian Department of Universities and Information Society (AGAUR) and the European Social Funds.

The authors wish to thank the reviewers for their constructive comments, Jordi Caubet, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose and Jaime Moreno from IBM and J.M. Cela, Albert Farres, Harald Servat and German Llort from BSC for their technical support.

REFERENCES

[1] http://www.top500.org/lists/2007/06.

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

[3] D. Alpert. Will microprocessor become simpler? *Microprocessor Report*, Nov 2003.

[4] 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.

[5] D.C. Bossen, J.M. Tendler, and K. Reick. Power4 system design for high reliability. *IEEE Micro*, 2002.

[6] F.J. Cazorla, P.M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in smt processors: Synergy between the os and smts. *IEEE Trans. Comput.*, 55(7):785–799, 2006.

[7] O. Celebioglu, A. Saify, T. Leng, J. Hsieh, V. Mashayekhi, and R. Rooholamini. The performance impact of computational efficiency on HPC clusters with hyper-threading technology. In *18th International Parallel and Dis tributed Processing Symposium (IPDPS'04) - Workshop 14*, 2004.

[8] M. Curtis-Maury, T. Wang, C. Antonopoulos, and D. Nikolopoulos. Integrating multiple forms of multithreaded execution on multi-smt systems: A study with scientific applications. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pages 199–, 2005.

[9] 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.

[10] 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.

[11] 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.

[12] 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.*

[13] IBM. Cell broadband engine architecture.

[14] IBM. Cell broadband engine programming handbook.

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

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

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

[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] D. T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), Feb 2002.

[23] M. Moreto, F.J. Cazorla, A. Ramirez, and M. Valero. Mlp-aware dynamic cache partitioning. In *International Conference on HiPEAC*, 2008.

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

[25] 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.

[26] M.K. Qureshi and Y.N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

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

[28] M.J. Serrano, R. Wood, and M. Nemirovsky. A study of multistreamed superscalar processors. Technical Report #93-05, University of California, Santa Barbara, 1993.

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

[30] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[31] 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.

[32] 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.

[33] D.M. Tullsen, S. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

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