Countdown Slack: A Run-Time Library to Reduce Energy Footprint in Large-Scale MPI Applications

(Article begins on next page)

08 July 2024

COUNTDOWN Slack: a Run-time Library to Reduce Energy Footprint in Large-scale MPI Applications

Daniele Cesarini, Andrea Bartolini, *Member, IEEE,* Andrea Borghesi,
Carlo Cavazzoni, Mathieu Luisier, *Member, IEEE,* and Luca Benini, *Fellow, IEEE*

✦

Abstract—The power consumption of supercomputers is a major challenge for system owners, users, and society. It limits the capacity of system installations, it requires large cooling infrastructures, and it is the cause of a large carbon footprint. Reducing power during application execution without changing the application source code or increasing time-to-completion is highly desirable in real-life high-performance computing scenarios.

The power management run-time frameworks proposed in the last decade are based on the assumption that the duration of communication and application phases in an MPI application can be predicted and used at run-time to trade-off communication slack with power consumption. In this manuscript, we first show that this assumption is too general and leads to mispredictions, slowing down applications, thereby jeopardizing the claimed benefits. We then propose a new approach based on (i) the separation of communication phases and slack during MPI calls and (ii) a timeout algorithm to cope with the hardware power management latency, which jointly makes it possible to achieve performance-neutral power saving in MPI applications without requiring labor-intensive and risky application source code modifications. We validate our approach in a tier-1 production environment with widely adopted scientific applications. Our approach has a time-to-completion overhead lower than 1%, while it successfully exploits slack in communication phases to achieve an average energy saving of 10%. If we focus on a large-scale application runs, the proposed approach achieves 22% energy saving with an overhead of only 0.4%. With respect to state-of-the-art approaches, *COUNTDOWN Slack* is the only that always leads to an energy saving with negligible overhead ($< 3\%$).

Index Terms—HPC, MPI, DVFS, power management, reactive policy.

- D. Cesarini and C. Cavazzoni are with the Department of SuperComputing Applications and Innovation (SCAI), CINECA, 40033 Casalecchio di Reno (BO), Italy (e-mail: d.cesarini@cineca.it;c.cavazzoni@cineca.it).
- A. Bartolini is with the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi", University of Bologna, 40136 Bologna, Italy (e-mail: a.bartolini@unibo.it).
- A. Borghesi is with the Department of Computer Science and Engineering, University of Bologna, 40136 Bologna, Italy (e-mail: andrea.borghesi3@unibo.it).
- M.Luisier is with the Department of Information Technology and Electrical Engineering, Swiss Federal Institute of Technology in Zurich, 8092 Zurich, Switzerland (e-mail: mluisier@iis.ee.ethz.ch).
- L. Benini is with the Department of Information Technology and Electrical Engineering, Swiss Federal Institute of Technology in Zurich, 8092 Zurich, Switzerland and the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi", University of Bologna, 40136 Bologna, Italy (e-mail: lbenini@iis.ee.ethz.ch).

1 INTRODUCTION

With the end of Dennard's scaling [1], [2], the last decade has seen a progressive increase of the power density required to operate each new processor generation at its maximum performance. Supercomputing installations have suffered from this power density increase, which over the years has pushed up the energy provisioning and cooling costs. While more efficient cooling techniques have been adopted to reduce the energy wasted at the infrastructure level, e.g. hot-water and free cooling [3], [4], [5], and more specialized computing elements with a higher ratio of vector and SIMD units with respect to general-purpose processors have emerged [6], [7], but a lot remains to be done in practice to reduce the energy wasted during computation.

Processor designers have addressed this aspect by embedding in their products finer and smarter power management support to automatically trade off performance for power consumption [8], [9]. By mission and design, the high-performance and scientific computing sectors aim at maximizing the peak performance of the computing systems, hence these techniques are seen as detrimental to the time-to-solution and time-to-science, and often disabled [10].

Indeed, low power design strategies enable computing resources to trade-off their performance for power consumption by means of low-power modes of operation. These power states are obtained by Dynamic and Voltage Frequency Scaling (DVFS) (also known as performance states or P-states [11]), clock gating or throttling states (T-states), and idle states which switch off unused resources (C-states [11]). While the built-in hardware and operating system (OS) policies are application-agnostic, in recent years several approaches have been proposed to let the final user control them in userspace [12], [13], [14], [15] and at execution time [16], [17].

The first family of approaches intends to trade-off power consumption and performance to gain energy efficiency [13], [14], [18], [19]. These techniques explore the use of HW power management knobs and application parameters to study the execution time (Time-to-Solution, TtS), average power, and energy (energy-to-solution, EtS) dependency

with respect to these knobs and parameters. While these approaches can be used in combination with autotuners and resource management frameworks to explore the EtS-TtS Pareto curve, they have a limited potential in the current supercomputing scenario: slowing down applications is almost always detrimental to the total cost of ownership (TCO) due to the large contribution related to the depreciation cost of the IT equipment [20].

The second family of approaches focuses on improving application performance under a power cap [15], [21], [22]. These approaches target power limited systems, computing nodes, and processing elements. They rely on the runtime capability of tracking the critical task in the application; then, the power budget of the node/socket/core running the critical task is dynamically relaxed while tightening the power budget of the non-critical resources. This not only involves software approaches [21], but also HW power management solutions, like Intel Turbo mode, and RAPL [23]. These methods are tailored to power capped supercomputing systems that still belong to a niche [24].

The third and last family of approaches aim at cutting the IT energy waste by reducing the performance of the processing elements when the application is in a phase with communication slack available [16], [22], [25], [26], [27], [28], [29], [30], [31]. These approaches try to isolate at runtime regions of the application execution flow which can be executed at a reduced P-state without impacting the application performance. While the hardware power management logic in today's processing elements is effective in reducing the power consumption of idle resources, in large-scale MPI parallel applications that fully utilize all the assigned processing elements workload unbalance, synchronization, and communication slack can be exploited to save energy. Several works have been proposed to address this challenge. However, also for these approaches slowing down the application is detrimental for the TCO thus making performance-neutral approaches more appealing.

The power management run-time frameworks which have been proposed in the latter family are based on the assumption that the duration of communication and computation phases in an MPI application can be predicted at execution time. In this manuscript, we first show that this assumption is too optimistic and leads to mispredictions, slowing down the application execution time, which jeopardizes their benefits. We then propose *COUNTDOWN Slack*[1] a new approach based on (i) the separation of communication phases and slack during MPI calls and (ii) a timeout algorithm to cope with the hardware power management latency, which jointly allows us to achieve performance-neutral power saving in MPI applications. We validate our approach in a tier-1 production environment with a widely adopted scientific benchmark suite [32], and a two-times ACM Gordon Bell Prize finalist application [33], [34]. We also compared *COUNTDOWN Slack* with the main state-of-the-art approaches. In average *COUNTDOWN Slack* reduces the energy consumption of 9.96% with an average overhead of 0.79%. When compared with the state of the art, *COUNTDOWN Slack* is capable of achieving similar energy saving but with negligible impact on the application performance.

1. Github Repository: https://github.com/EEESlab/countdown

If we consider the worst-case performance degradation, *COUNTDOWN Slack* has a minimal impact on performance, just 3.02% overhead, while the worst-case overhead for state-of-the-art approaches is between 8.92% and 144.75%. If we consider that only a negligible overhead (below 5%) is acceptable, *COUNTDOWN Slack* is the only approach that never exceeds this value for all the applications while at the same time always leading to an energy saving. In contrast, state-of-the-art approaches can cause non-negligible overheads or severe energy losses. Worst-case energy saving for *COUNTDOWN Slack* is 1%, while for the state-of-the-art approaches it ranges from 0.05% to -24.69%.

The paper is organized as follows. Section 2, presents the state of the art in power and energy management approaches for HPC computing systems. Section 3 introduces a background of power-saving in MPI-based applications. Section 4 describes the implementation of our *COUNTDOWN Slack* runtime. Section 5 explains our implementation of the state of the art of the energy-aware runtime that we use to compare with *COUNTDOWN Slack*. In Section 6, we report an analysis of our benchmarks in terms of predictability of computation and communication region of the application. Moreover, we report experimental results in terms of overhead, energy and power saving for production applications in a tier-1 supercomputer.

2 RELATED WORK

Energy Efficiency is a hot topic in supercomputing and several power management approaches have been developed to reduce the power impact in HPC systems. But implementing energy-efficient strategies requires also software interfaces with the hardware to collect data on energy and power consumption, and to control the performance of the system.

Intel processors have a plethora of features for power and energy measurement and control. Accessing these features requires "ring 0" access, which in practice means working within the system kernel calls that can access to privileged MSRs usually forbidden to standard users. The MSR-SAFE driver [35] overcomes this limitation allowing the access only on a subset of the MSRs within a strict control of system admins which can define a fine-grain whitelist for allowing access to privileged hardware registers.

Machine specific registers allow measuring energy and power at the chip level. To extend these measurements with fine-grain and accurate node-level ones, additional hardware is required to be installed at the motherboard level. HDEEM [36] developed by BULL, is an FPGA card integrated into the node which is capable of sampling the power consumption of the entire node and main components and exposing this data to the BMC and O.S..

Similar to HDEEM, DIG [37] is able to sample the energy and power of an HPC compute node at a very high resolution, up to 20μ with an accuracy of 1%. While HDEEM is capable of a time resolution of 1 ms at the power plug and 10 ms on voltage-regulators (CPU, DDR), with an accuracy of 2% and 5%, respectively. DIG is integrated with a holistic monitoring framework and capable of real-time edge computing analysis [38]
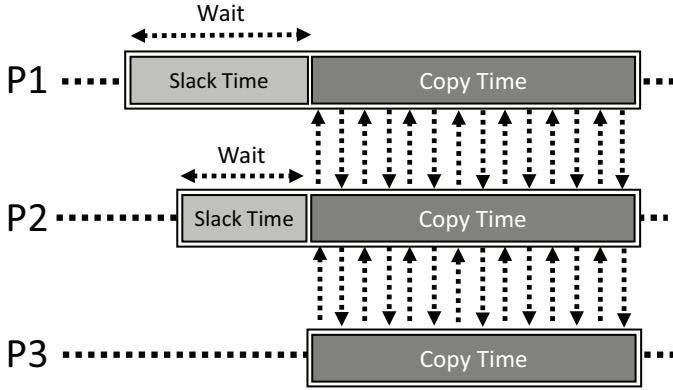
Fig. 1: Execution model and taxonomy. A task is shown for each process. Where the sum of Slack Time (*Tslack*) and Copy Time (*Tcopy*) is equal to MPI Communication Time (*Tcomm*).

Amester [39] developed by IBM is a software tool to remotely collect power, thermal, and performance metrics from IBM servers. It can connect to OpenPOWER systems by accessing the board management controller (BMC). In this way, it can be used to connect to several systems simultaneously and collect the monitored metrics. It is specific to IBM servers.

PowerAPI [40] provides a simple and standardized way to estimate the energy consumption of a given process through a set of APIs. PowerAPI has been designed to provide a single interface for accessing these quantities limiting the proliferation of vendor/OEM-specific interfaces [35], [36], [37], [39].

In this work, we will rely on MSR-SAFE tool to monitor and control the power management on the system and to collect hardware performance counters bevause our target system does not have hardware monitor boards and we need access to energy/power sensors and hardware performance counters, respectively.

In the last decade, several works were proposed to reduce the energy waste of large scale MPI applications. Kappiah et al. [22] show that in an MPI parallel application it is possible to use the PMPI profiling interface [41] to intercept MPI calls and isolate the time spent by each rank during an application iteration in communication (*Tcomm* in Figure 1) and computation. The authors show that in collective MPI primitives the amount of synchronization slack (*Tslack* in Figure 1) can be converted into power (and energy) reduction by slowing down the computation (lowering the processor's P-state) to absorb the available slack. This operation can be done under the assumption that the communication and computation time of the upcoming application iteration can be known upfront. The authors of the paper propose to use an error signal (desired vs measured slack) computed on the previous iteration to drive the P-state reduction. Iterations need to be marked by the user.

Lim et at. [26] show that during the communication time (time spent in the MPI library) of an MPI parallel application the cores' P-state can be significantly reduced without causing severe overheads; thus they propose to execute the communication phases at a reduced P-state and computing phases at the default one. Due to the latency of P-state transitions, it is not feasible to target all the communication phases singularly, but the authors propose to group them based on a proximity index and allow a P-state reduction only in regions of the code with higher communication density. Regions of the code are constructed and marked at execution time by leveraging the hash of the call stack to generate a *Task id* when MPI primitives are encountered. The proposed algorithm uses the last value prediction to determine the beginning and the end of a given region or the P-state to be applied for the upcoming region. The P-state is selected based on the measured IPS (instructions per second) on the previous region and a pre-characterization of the optimal P-state for a given IPS level.

Sundriyal et al. [42], [43], [44] focus on the All-to-All [43], send/receive [42], and AllGather communications [44]. They analyze the impact of fine-grain power management strategies in MVAPICH2 communication primitives (considered singularly) and their results suggest that different regions in the MPI primitives have different power/performance trade-offs.

Rountree et al. [25] propose to separate the communication time into the slack and copy time. The slack time is caused by waiting for the critical task to enter the MPI primitive, and the data transfer causes the copy time. The authors define the task as the region of code between two MPI communication calls and define an optimization problem to minimize the slack time and save power (and energy) by reducing the P-state of the core during computation time. In [16] the same authors propose three online algorithms *Fermata*, *Andante*, and *Adagio* (described in Section 5) that use, similarly to [26], the hash of the call stack (*Task id*) at the entrance of an MPI call to identify similar tasks that are used in prediction algorithms to identify energy hotspots. It must be noted that the approaches used in [16], [25], [26] will potentially lead to misprediction inducing costly performance overhead or loss of energy-saving opportunities, which can also become severe in irregular applications [27].

The exploration work of Dosanjh et al. [45] shows a quantitative comparison of the performance and power consumption between onloaded and offloaded network stack processing for systems that use CPU frequency variations. The experimental results clearly illustrate the potential benefits of network offloading compared to onloaded at low frequencies. While offloading is very effective in power saving and performance loss at low frequencies, respectively up to 30.5% and 1.5%, onloaded network adapter shows a huge performance loss at low frequencies that can reach up to 35.1% of performance degradation. Our target HPC system is equipped with offloaded network adapters.

Bhalachandra et al. [28], as in [16], [25], focus on saving power by entering a low power state for processes which are not in the critical path. The authors propose an algorithm to save energy by reducing application unbalance. This is based on measuring the start and end time of each MPI_barrier and MPI_Allreduce primitives to compute the duration of application and MPI code. Based on that, the authors propose a feedback loop to lower the P-state and T-state if in previous computation and MPI regions the overhead was below a given threshold. This algorithm is based on the assumption that the duration of the current

application and MPI phases will be the same as the previous ones.

Eastep et al. propose GEOPM [15], an extensible and plug-in based framework for power management in large parallel systems. GEOPM is an open-source project and exposes a set of APIs that programmers can insert into applications to combine power management strategies and HPC workload. A plugin of the framework targets power constraint systems aiming to speed up the critical path giving a larger power budget to the CPU's executing the critical path tasks. In a similar manner, another plugin can selectively reduce the frequency of the processors in specific regions of codes flagged by the user by differentiating regions in CPU, memory, IO, or disk bound. Today, GEOPM is capable of identifying MPI regions and reducing the frequency based on MPI primitive type. However, it cannot differentiate between short and long MPI and thus cannot control the overhead caused by the frequency changes and runtime in short MPI primitives. *COUNTDOWN Slack* addresses this limitation and can be integrated into future releases of GEOPM, as its design principles are entirely compatible with it (i.e. no application code modifications are required).

Venkatesh et al. [29] show that the approaches based on temporal execution patterns for predicting slack (such as last value prediction) [16], [25], [26], [46] can lead to significant misprediction errors. The authors propose to use a combination of empirical observation and communication models specialized for the different classes of communication primitives for estimating the duration of the MPI phases. If this estimation is long enough, they will decide to reduce the P-state.

Cesarini et al. [30], [31] tries to overcome mispredictions by leveraging a timeout policy (namely COUNTDOWN) which sets a timer at the entrance of any MPI calls to discard communication times shorter than the hardware power controller latency, which is measured to be $500\mu s$, in line with the findings of Hackenberg et al. [8]. If the communication time of the MPI primitive is longer than $500\mu s$ when the timer callback is triggered the lowest P-state is selected, otherwise the timer is canceled. This timeout-based policy has been in-depth analyzed in the power management literature and has been shown to be effective in mitigating the issues related to prediction inaccuracy and predictive model overfitting [47]. The *COUNTDOWN* approach does not distinguish between slack and copy time and execute both of them at the minimum P-state, causing additional overheads.

Hence, in this work, we propose *COUNTDOWN Slack*, a novel technique that, while inspired by the methodology proposed in [30], [31], differently from state-of-the-art approaches induces negligible overhead (in the worst case less than 3%) in applications running on real production HPC machines. *COUNTDOWN Slack* implements purely reactive mechanisms, thus it is robust to miss-prediction errors and capable of isolating slack time with a new reactive approach based on artificial barriers insertion. We will discuss these aspects in Sections 3 and 4.

3 BACKGROUND

The proposed manuscript shares some common assumptions with approaches in the state of the art. We list in this Section common definitions, as well as a taxonomy to compare our solution with previous works in this field.

3.1 Definitions and Assumptions

We target typical HPC scientific applications, which are usually composed of parallel processes (from tens to thousands) running on a cluster of compute nodes interconnected with a high-bandwidth low-latency network. Each application process is statically bound to a compute element for its entire life duration. Processes can exchange data through the network interconnection using a message-passing interface (MPI) library that can send explicit messages. Multiple processes can share the same compute node since modern HPC machines are equipped with multi- and many-core high-end processors. MPI library abstracts the locality of computing resources by taking care of the communication inter and intra nodes. HPC applications can ignore the locality of the processes because MPI represents the computation resources as a large set of single-core nodes. HPC users can choose the binding configuration of the processes to optimize communication. Non-uniform memory access (NUMA) plays an essential role in terms of communication latency and bandwidth in MPI communication primitives.

In this work, we do not consider heterogeneous compute nodes such as GPU- and MIC-based architectures, leaving them to future extensions of this work. We recall that today (Nov.18 Top500 list [6]) 73% (364/500) of worldwide supercomputers systems are homogeneous x86 based clusters.

Figure 1 shows our execution model and taxonomy, which is the same of Rountree et Al. [16]. At the base of the execution model, there is the *task*; each process is composed of a set of tasks that sequentially enter in the execution flow. Each process starts with the MPI primitive *MPI_Init*, which begins the first task of the process, and ends with the *MPI_Finalize*, which concludes the last task. Each task comprises (i) a computation time *Tcomp*, which identifies the time spent in the application code, and (ii) a communication time *Tcomm*, which is the time spent in the MPI library. In turn, *Tcomm* is composed of (i) a slack time *Tslack* and (ii) a copy time *Tcopy*. The slack time is the period spent in the MPI library while waiting for the last process that encounters the MPI primitive. This time is purely busy waiting and is equal to zero for the last process. When the last process enters in the primitive, it unlocks all the others. When *Tslack* is concluded, processes can start sending and receiving data – we call this period copy time. The MPI specification also implements pure synchronization primitives (such as *MPI_Barrier*, *MPI_Wait*, etc.) which have zero copy time but cause slack time. In this work, we do not target non-blocking and no-synchronization MPI primitives since they do not produce busy waiting wasted cycles in the application. We define the *critical process* as the last process that enters the MPI primitive; this process plays a critical role as it blocks all the others. A *critical path* is a list of *critical processes* in successive blocking MPI primitives.

3.2 Power Management Basics

Today's generation of high-end CPUs is composed of many processing elements that can work at different voltage and frequency. In particular, Intel technology started to integrate

per-core fully-integrated voltage regulators from Haswell architecture [48], which allow the DVFS mechanism to trade-off performance and energy. While DVFS control is available in most modern high-end processors for HPC system, our approach explicitly targets Intel architectures for two reasons: i) our target machine is an Intel-based system and ii) most of HPC system in the Top500 list (73% in Nov 2018 [6]) are based on Intel CPUs.

The power control unit (PCU) of Intel architectures is the HW component that controls the power management knobs and exposes model-specific registers (MSR) concerning the DVFS control knobs. While the internal logic of Intel PCUs is not publicly available, Hackenberg et al. [8] analyzed the behavior of the DVFS control registers of the PCU in Haswell architectures. Their experimental results show that frequency changes occur at regular intervals of about $500\mu s$. As pointed out by Cesarini et al. [30], [31] this interval creates uncertainties in P-state transitions for code regions shorter than $500\mu s$.

It is possible to interact with the DVFS mechanism using the MSRs. MSRs are not only used to interact with the HW power manager, but also with the performance counters, debugging, and trace controls. *COUNTDOWN Slack* requires read and write access to these registers. Intel provides two specific assembly instructions to read and write MSRs, named respectively *rdmsr* and *wrmsr*. Both instructions are executed in ring 0 (kernel mode), so only the operating system (OS) can execute them. The Linux OS allows user-space access through a kernel driver called *MSR driver*. The drawback of the *MSR driver* is that only the root user can access to this driver because exposing all MSRs to a generic user can lead to security issues.

The *MSR_SAFE driver* [35] overcome the restricted privilege issue and security risks, by supporting a registers white-list. In *COUNTDOWN Slack* we white-listed a limited subset of control registers in [35] to let standard HPC users interact with the HW power manager and performance counters.

The *MSR_SAFE driver* also implement a batch mode to read the MSRs, this means that in a single system call it is possible to collect a large set of MSRs avoiding the overhead of multiple system calls. We rely on the batch mode to sample the compute node in a time fashion to collect data for power, energy and a plethora of performance counters. While we use the low-overhead user-space *RDPMC* instruction to access the performance monitoring units implemented in Intel PCU for fine-grain access to monitor frequency, the time stamp counter (TSC) and the instructions retired for each MPI call and application region.

3.3 Power Management Modelling

The majority of HPC power management policies fall into two categories, proactive and reactive policies. Both policies aim at scaling down the P-state in regions of code which are less susceptible (or even not sensitive at all) to frequency scaling. We now report the most common implementation concepts for both categories.

3.3.1 Proactive Policies

Training strategies are often at the base of proactive policies. A typical training strategy initially starts by identifying the portion of the code that can be targeted for frequency reduction. The runtime always executes a newly encountered code region at the highest available P-state to measure the performance of that region. At the end of the code portion, the runtime stops the performance monitor and gathers the performance parameters; these will be used by the algorithm to compute a new P-state for the next time the code region is encountered. The most frequently used parameters are the execution time of the code region at a given frequency and the number of retired instructions. The next time the execution flow encounters a previously recorded code region, the policy tries to scale the P-state, by applying the earlier computed P-state. A code region can be any part of the execution flow of the program, such as application code [16] or communication runtime [26]. The algorithms used to identify the optimal frequency can be a simple last-value prediction [16] or many complex predictors, like an auto-regressive moving average. A typical implementation of this policy is done through a history table used by the algorithm to predict the next P-state to be assigned. The code regions can be uniquely identified using different strategies: (i) source code instrumentation [22], (ii) compilers automatic insertion [49] or (iii) identified at execution time via a stack trace mechanism [16], [26]. The advantage of using source code tagging is the low overhead and precise code pinpoint that developers can optimize. However, this methodology requires to modify the application source code which is not always tolerated. On the contrary, compiler tagging does not require programmer intervention. The drawback is the need to re-compile the source code. Conversely, the stack trace mechanism is completely application-agnostic, and it does not require source code modification nor re-compilation, but extra cycles of computation in the application. As we will see in the experimental results when this is done synchronously to the MPI calls its overhead can be neglected.

3.3.2 Reactive Policies

Differently from the previous ones, reactive policies are implemented as event-based strategies. When specific events occur, the runtime triggers well-defined actions. Cesarini et al. [30], [31] developed *COUNTDOWN*, which implements a reactive policy based on a timeout to filter out code portions which have too short to cause a P-state transition, this depends on the HW PCU delay of Intel architectures. Contrary to proactive ones, reactive policies do not need to uniquely identify regions of code, since they do not maintain a history of the execution traces. However, similarly to the proactive ones, the runtime requires to intercept events related to the beginning and end of each region. This can be done by leveraging the profiling extensions of communication libraries such as PMPI or debug symbols [50].

While proactive policies can modify their behavior to adapt to different code regions and minimize overhead, reactive policies always apply the same operation, since they are unaware of the different sensitivities to the frequency scaling of different code portions. Similarly to *COUNTDOWN*, *COUNTDOWN Slack* applies a reactive policy to code regions without considering their sensitivity to frequency scaling.
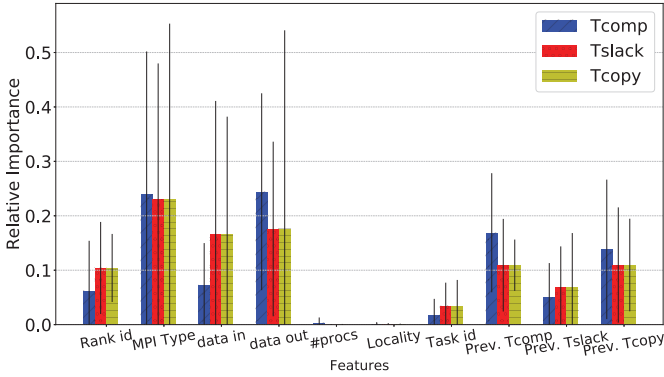
Fig. 2: Relative importance of each feature (including information about previous phases); the colors identify the prediction targets

Application	Without Previous Info			With Previous Info		
	Tcomp	Tslack	Tcopy	Tcomp	Tslack	Tcopy
nas_bt.E.1024	57.0	17.6	52.5	6.2	12.4	12.4
nas_cg.E.1024	21.9	7.1	25.3	16.2	5.5	11.0
nas_ep.E.128	9.1	8.4	23.8	9.7	7.3	24.6
nas_ft.E.1024	1.2	5.4	9.7	0.3	1.2	3.9
nas_is.D.128	10.7	15.2	8.2	5.3	8.0	2.4
nas_lu.E.1024	0.9	19.8	0.5	0.7	13.5	0.4
nas_mg.E.128	5.1	4.8	13.0	4.1	5.3	13.1
nas_sp.E.1024	46.5	11.8	46.9	4.1	10.2	7.3
omen_1056p	1.0	57.3	75.8	2.8	55.4	64.6
Average	17.0	16.4	28.4	5.5	13.2	15.5

TABLE 1: Prediction error [%] for all test applications (SMAPE)

3.4 Regions Predictability in Proactive Policies

As depicted in Section 3.3.1 most of the state-of-the-art energy-management runtimes use a predictor to estimate the duration of the regions of an application for optimization purposes. We report an analysis carried out on the test applications to highlight the degree of predictability of the region *Tcomp*, *Tslack*, and *Tcopy*, also shown in Figure 2, based on features available before a given region is encountered. To assess the predictability of the region duration, we employ a standard Machine Learning (ML) technique, namely Random Forest (RF) [51] models[2]. For each test application, we build and train three RF models, one for each target region duration (*Tcomp*, *Tslack*, and *Tcopy*); we then evaluate the quality of the prediction by measuring the difference between the real region duration and the estimates (over a test set of examples not seen during the training phase). All the experiments on the predictability were performed using the *scikit-learn* [52], a widespread ML library for python.

For this purpose we extracted with *COUNTDOWN Slack Event Profiler* a set of traces for the NPB and OMEN benchmarks in the default configuration (*Baseline*). For each benchmark (8 NPB + OMEN) we obtain a trace with a row for each code region and a column for each feature analyzed. The rows are ordered first on the rank id, then on the task id, and finally on the progression of time.

We consider two types of approach, namely a first one taking into consideration only the features relative to the MPI call whose region duration we want to predict, and a second one that exploits also information about the last MPI call of the same type and rank. We now describe the first approach. Each data set is composed of the following features: 1) *Rank id*: the rank of the MPI process that makes the MPI call; 2) *MPI Type*: type of the MPI call; 3) *data in*: size in byte of the received data; 4) *data out*: size of the sent data (byte); 5) *#procs*: number of processes involved in the MPI call; 6) *Locality*: a number between 0 and 1 that specifies the amount of local (to the node) or remote processes involved in the call (0 means that all processes are remote, 1 all local processes); 7) *Task id*: the hash of the stack (identify MPI calls made at the same points of code). Each

RF model aims at predicting one of the three targets, *Tcomp*, *Tslack*, and *Tcopy* (expressed in microseconds). We limit our analysis to processes with a duration longer than 500ms, we are less interested in extremely short MPI calls since they offer lower potential in terms of power management. There is a distinct data set for each test application, with sizes of varying dimensions, ranging from 1K elements to 1 million examples. We recall that the first approach is motivated by the state-of-the-art works and uses pre-characterization of the copy time to estimate at execution time the slack time. Indeed the *Tcopy* model can be seen as a generalization of the pre-characterization steps.

The second approach is motivated by state-of-the-art methods that exploit information regarding previous MPI calls to estimate the phase duration (last value prediction). More precisely, in this case, each example in the data set possesses the set of aforementioned features plus three more fields: the *Prev. Tcomp*, *Prev. Tslack*, and *Prev. Tcopy* values of the last MPI call with the same type, task, and rank. In both approaches (with or without previous MPI call information) the data set was split between the training set and the test set; the split ratio was 70% of the data to the training set (used to train the ML model) and 30% to the test set (used to evaluate the quality of the predictions). After a preliminary analysis, we discovered that the RF model accuracy increased if we used the natural logarithm of the target during the training phase, rather than directly using the duration in microseconds; probably, this happens because the logarithm flattens the peaks caused by extremely long or short duration (with respect to the average phase duration for the training set). The accuracy results reported later (test phase) are instead computed on the actual values, by applying the exponential function to the predictions made by the RF models.

In Table 1 we see the results of the predictability experiments. The prediction error is computed as percentage, in particular using the SMAPE metric (Symmetric Mean Absolute Percentage Error), preferred to the standard mean absolute percentage error since the former places smaller emphasis on regions with shorter duration[3]. Each row corresponds to a different test application (identified by the first column); the final one reports the average over all applica-

---

2. Random Forests are ensembles of decision trees

3. For a single prediction it is computed as: $SMAPE = 100 \cdot \frac{abs(pred - actual)}{pred + actual}$, where $pred$ is the predicted region duration and $actual$ is the real value. If we use only the actual value at the numerator, as in the standard mean average percentage error, examples with very short duration would significantly skew the overall error

tions. Columns 2-4 show the error obtained if we train ML model without providing information about the duration of previous regions; columns 5-7 report the results obtained *with* previous region information. The pairs of three columns *Tcomp*, *Tslack*, and *Tcopy* correspond to the three region-targets, one triplet for each approach (with/without previous information).

Table 1 reveals that it is not straightforward to predict the target phase duration with the collected information. If we consider the case without previous MPI call information and we look at the average values computed over all applications, the error for predicting both *Tcomp* and *Tslack* is around 16-17%, while for *Tcopy* the accuracy is even lower (28% error). As one could have expected, things improve if we feed the ML models with additional information regarding the last MPI calls of the same rank, task, and type; this is especially true for *Tcomp* (error decreased at around 5%) and *Tcopy*, while the improvement for *Tslack* is less significant. Generally speaking, considering previous information improves the prediction accuracy on most test applications (even drastically: see for example *nas_bt.E.1024* and *nas_sp.E.1024*). However, this is not true for all applications and in some cases (for instance *omen_1056p*) the additional information can lead to a marginal decrease in prediction accuracy, probably due to an increase in noise that confuses the RF models (however, the effect is marginal).

Another aspect that deserves more analysis is understanding the factors that the RF models focus on in order to make their estimates. We can gain some insight into this matter by inspecting the importance of each feature. In *scikit-learn* the feature importance is computed as mean decrease in impurity, described in [53]; this method is known to be prone to bias (see [54]), thus we opted to compute the feature importance via a permutation-based approach [55], [56], where each feature importance is computed by looking at how random re-shuffling (which preserves the feature distribution) influences the model accuracy. We normalized the important values in the [0,1] range; zero indicates that a feature has no importance for the regression RF model, while values closer to 1 indicate relatively more important features.

Figure 2 shows the importance of all features when we include information about previous MPI calls. The plot refers to the average feature importance computed overall test applications. The height of each bar corresponds to the average feature importance and the black vertical line represents the standard deviation – longer black lines indicate that the feature importance varies significantly for different applications. Each targetis highlighted by a different color, blue for *Tcomp*, red for *Tslack*, green for *Tcopy*. Each group of three bars represent a feature; the three right-most groups report the influence of the previous MPI call (same type) regions duration, respectively (from left to right), last MPI call *Tcomp*, *Tslack*, and *Tcopy*.

The first thing that we can notice is that the standard deviations tend to be quite large, revealing a high variability. This high variability suggests that predicting the region duration of MPI calls in HPC application is indeed a difficult task since there is no subset of trustworthy features that can be robustly employed for accurate estimation. Secondly, the number of processes, the locality, and the task id have very

little influence on the RF prediction. We recall, that the task id was one of the foremost important feature in state-of-the-art approaches [16], [26].

To summarize, the features with greater importance are the size of the outgoing transmitted data (especially for the *Tcomp*) and the type of the MPI call. The size of the incoming data is more relevant for *Tslack* and *Tcopy*. Additionally, the importance of the information about the previous MPI call with the same type cannot be discounted, as highlighted by the right-most three groups of columns. In particular, the length of *Tcomp* and *Tcopy* phases of the last MPI call is an important factor for the models that predict *Tcomp*. For these reasons, *COUNTDOWN Slack* employs only reactive mechanisms to isolate slack and filter out short slack regions, which avoid miss-prediction errors.

4 COUNTDOWN SLACK - A LOW-OVERHEAD, REACTIVE, SLACK-AWARE PM RUNTIME

In this Section we present the implementation of *COUNTDOWN Slack*.

4.1 Runtime

COUNTDOWN Slack is a simple shared library written in C language. It can instrument standard MPI-based applications that load *COUNTDOWN Slack* in their *LD_PRELOAD* environment variable before the execution of the program. Using this technique, every MPI call is intercepted by *COUNTDOWN Slack* which executes between the application and the MPI library. It implements a PMPI interface to wrap all the MPI primitives defined in the MPI specification v3. The library has been designed to have the lowest possible overhead and to interact with the hardware through the *MSR_SAFE* kernel driver as discussed in Section 3.2. *COUNTDOWN Slack* also provides a static link version, which can be used when dynamic linking is not possible, to inject *COUNTDOWN Slack* in the application binary at compilation time. If dynamic linking is allowed, *COUNTDOWN Slack* does not require any modifications of the source code nor toolchain, nor re-compilation steps. It is completely transparent to the user. In the experimental results of this paper, we instrument all the target HPC benchmarks using dynamic linking.

COUNTDOWN Slack is based on a simple but effective strategy to reduce energy consumption in production HPC systems without performance penalties. The key idea is to scale down the P-state in slack times of the application reducing the frequency but leaving unaltered the performance for both computation and data copy regions.

Since *COUNTDOWN Slack* targets performance-neutral energy savings, our goal is to avoid performance penalties for a large set of MPI-based applications, thus *COUNTDOWN Slack* focuses on saving energy only when this has no effects on performance skipping potential energy saving if they could induce non-negligible performance overhead. As side effect, *COUNTDOWN Slack* shows slightly lower energy saving respect to the state-of-the-art approaches but guarantees better performance which is the first goal in HPC applications. To accomplish this task, *COUNTDOWN Slack* employs a purely reactive policy for both slack isolation and
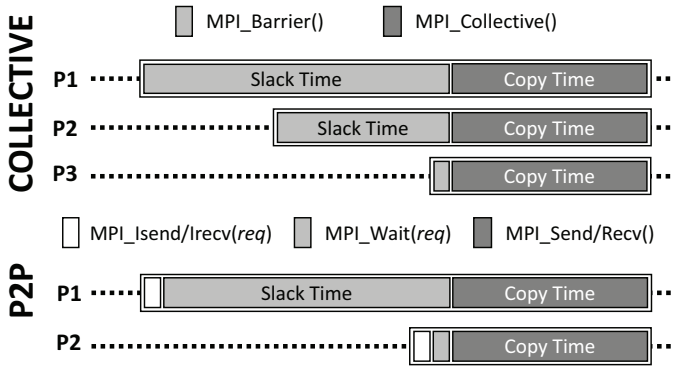
Fig. 3: Artificial barrier mechanism to split slack and copy time of an MPI primitive.

short region filtering since predictive algorithms can induce performance penalties in case of mispredictions [27].

As shown in [8], modern Intel architectures allow core frequency changes only every 500us. Thus, we implement a timeout policy as presented in [30], [31], but we apply it only to slack times, without varying the cores' frequency during the copy, as shown in Figure 4.

4.2 Reactive Slack Isolation

The slack time of an MPI call is usually included in the primitive. Differently from previous works that use pre-characterization or non-causal models to separate slack and copy time from the communication time [16], [25], [26], in *COUNTDOWN Slack* we propose a novel reactive approach based on the insertion of artificial/instrumental barriers as show in figure 3. This mechanism is agnostic to the MPI implementation, and it is built on top of standard MPI primitives. It can be used with every MPI library. We applied this mechanism on blocking MPI primitives leaving unaltered non-blocking, one-sided, file and support MPI primitives. We also account for collective and P2P (Point-to-Point) primitives since the time spent in other primitives are negligible for the considered benchmarks. We implement two different mechanisms to isolate the slack time, one for collective and one for P2P primitives.

4.2.1 Collective Barrier

We propose a straightforward mechanism to separate the slack (*Tslack*) and the copy (*Tcopy*) time in collective primitives. This mechanism is depicted in the top part of the figure 3. Every time that the application calls a collective primitive, *COUNTDOWN Slack* intercepts the call and forces an artificial *MPI_Barrier* on the same communicator. During the barrier, *COUNTDOWN Slack* can reduce the P-state to the minimum available in the system if the *MPI_Barrier* is longer than a specified time threshold (Reactive Short Phase Filter). When all the processes encounter the collective primitive, the barrier terminates and *COUNTDOWN Slack* restores the maximum frequency profiling the slack time. After that, the execution flow returns to the application which can call the real collecting primitive. We call this mechanism *Collective COUNTDOWN Slack barrier*.

4.2.2 Point-to-Point Barrier

The mechanism for collective barriers is respected since all the processes in the communicator are involved in the barrier. Unfortunately, the P2P primitives are called only from the processes involved in the P2P communication. We cannot insert a *MPI_Barrier* because the following *MPI_Wait* would cause a deadlock on the processes involved in the P2P communication. To overcome this limitation, we implemented a waiting mechanism based on non-blocking primitives shown in the bottom part of the figure 3. Before a *MPI_Send* primitive *COUNTDOWN Slack* adds an artificial *MPI_Isend* followed by a *MPI_Wait*. Similarly before a *MPI_Recv* primitive *COUNTDOWN Slack* adds an artificial *MPI_Irecv* followed by a *MPI_Wait*. Differently, before a *MPI_Isend* primitive *COUNTDOWN Slack* adds only an artificial *MPI_Isend*, and before a *MPI_Irecv* primitive *COUNT-DOWN Slack* adds only an artificial *MPI_Irecv*.

The non-blocking P2P primitive returns back a request object that *COUNTDOWN Slack* uses in the following *MPI_Wait* primitive. The *MPI_Wait* is a blocking primitive used to wait for the completion of a request object. When the application enters in the artificially *MPI_Wait*, *COUNT-DOWN Slack* reduces the frequency if the *MPI_Wait* is long enough (Reactive Short Phase Filter). This mechanism allows *COUNTDOWN Slack* to obtain a P2P barrier just between the processes involved in the P2P communication and to isolate its slack to copy time. We call this mechanism *P2P COUNTDOWN Slack barrier*. When all processes reach the P2P primitive, the artificial barrier terminates and the execution flow returns to *COUNTDOWN Slack*. After that, the library restores the maximum frequency and calls the original P2P primitive of the application.

COUNTDOWN Slack instruments blocking P2P primitives with the correspondent non-blocking ones but the application can use mixed blocking and non-blocking P2P primitives and this can create a mismatch of *P2P COUNT-DOWN Slack barrier*. To avoid it, we added non-blocking P2P primitives in-front-of every non-blocking P2P primitives called from the application to balance the number of non-blocking P2P primitives.

To measure the overhead for both *collective and P2P COUNTDOWN Slack barrier*, we run all our benchmarks with and without the artificial barrier mechanism and we compare the execution times. The experimental results show a negligible overhead for all our benchmarks.

4.2.3 Synchronization and Wait MPI Primitives

In MPI specification, there is a set of MPI primitives that are purely synchronization primitives (as instance *MPI_Barrier*) or used to wait for specific events (like *MPI_Wait MPI_Waitall*, *MPI_Waitany*, etc.). *COUNTDOWN Slack* excludes these specific MPI primitives from the logic used to extract the slack isolation because their MPI time is only composed to slack time. In this case, *COUNTDOWN Slack* directly applies its power management strategy without using the slack isolation mechanism.

4.3 Reactive Short Phase Filter

As we showed in 3.2, it is not possible to ensure P-state transitions in code regions shorter than $500\mu s$. To filter out slack
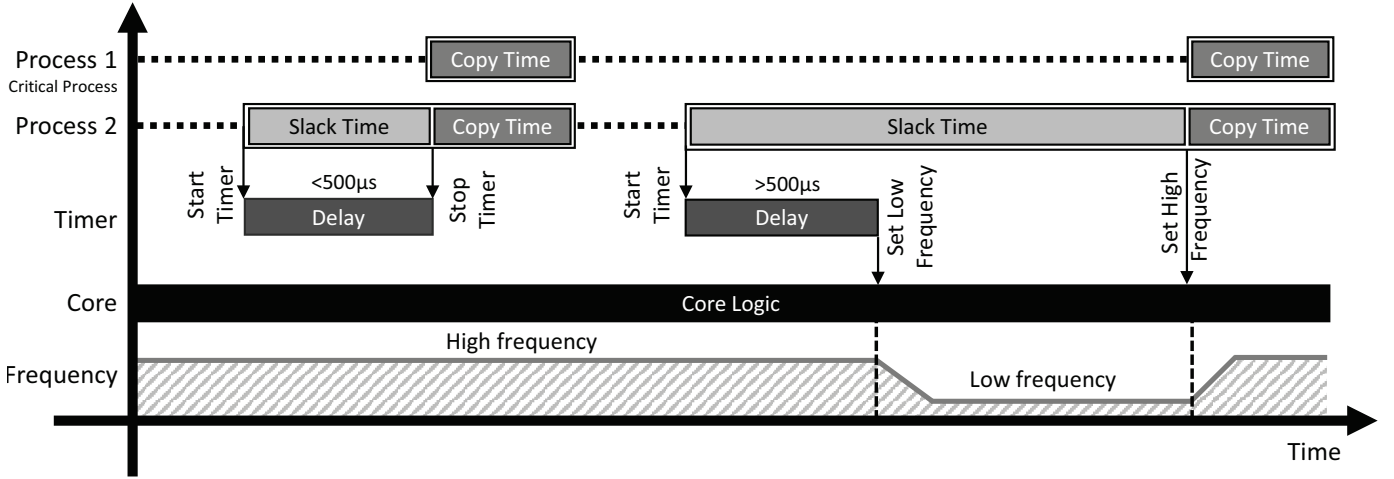
Fig. 4: Timer strategy utilized in *COUNTDOWN Slack* to filter-out short slack times.

regions shorter than this value, we implemented a timeout policy in *COUNTDOWN Slack* as the one proposed in [30], [31]. The timeout strategy of *COUNTDOWN Slack* relies on the standard timer APIs of Linux systems. Linux provides the kernel calls *setitimer()* and *getitimer()* to manipulate Linux timers. The timer allows users to register a callback function; when the timer expires, a system signal interrupts the "normal" execution, and the callback is executed. The callback sets the lowest P-state and return. At the end of the slack region, *COUNTDOWN Slack* restores the highest P-state. This mechanism is shown in Fig. 4.

COUNTDOWN Slack also implements a mechanism to exclude MPI primitives based on their type and on the size of the message sent/received to better tune the runtime avoiding source of excessive overhead and excluding special implementation of MPI primitives that can be optimized with non-blocking mechanisms. We do not use this exclusion mechanism in our experimental setup.

4.4 Profiler

COUNTDOWN Slack is endowed with a profiler module that allows a detailed analysis of the application; the profiler is split into two components.

i) The *event profiler* monitors the HW performance counters using the *RDPMC* instruction, reading the performance monitoring units of Intel's processors. *RDPMC* is a low-overhead user-space assembly instruction that can be used to keep track of micro-architectural events at a very high frequency with negligible overhead. This instruction reads the fix performance counters which counts the number of clock cycles at the nominal frequency, at the current P-state, and the instructions retired. Furthermore, it can read a limited subset of configurable HW performance counters used to monitor user-specific micro-architectural metrics. This profiler is also able to extract MPI information from the parameters passed to the MPI primitives.

ii) The *time-based profiler* collects a broad set of HW performance counters every second. This profile is time-based, and it leverages a timer to sample the entire node. Every second, the MPI processes on the same node alternately sample all the core and uncore performance registers in a round-robin fashion. This strategy is used to distribute the profiling overhead among all processes. The profiler maintains the tracing information in a memory area shared among all the processes. The profiler exploits the *MSR_SAFE* kernel driver to access the performance registers using the batch mode [57] to reduce the overhead. Moreover, it uses Intel Running Average Power Limit (RAPL) registers to monitor the energy/power consumed by the CPU and DRAM. The energy measurements presented in the rest of this work always refer to both package and DRAM consumption.

The profiler does not save all the events and time-based traces, but it summarizes them in a hierarchical report. This report comprises a summary file with information about the entire application run, an MPI report with information about MPI primitives, and a set of reports organized for nodes, sockets, and cores. These reports contain the same information of the summary report (plus specific metrics) but organized, respectively, for nodes, sockets, and cores. The hierarchical organization improves the readability of the reports and their long term compression. The overhead of the hierarchical report is entirely negligible, while the overhead of the event and time-based profilers are strictly related to the performance of the storage. In our target architecture with a small set of computing nodes (29), both event and time-based tracing overhead are negligible. The memory footprint of the profiler is constant due to the fixed number of performance counters. It is in the order of few megabytes per MPI process. We recall that the different profiler modalities can be easily configured and deactivated.

5 STATE-OF-THE-ART ENERGY-AWARE RUNTIMES

In this Section, we discuss the energy-aware runtimes that we have implemented as part of *COUNTDOWN Slack* for comparisons with the state of the art. In Section 6, we compare *COUNTDOWN Slack* with the following described algorithms.

5.1 Fermata

The first runtime we introduce for comparison with *COUNTDOWN Slack* is *Fermata* [16], [58]. *Fermata* imple-

ments a simple algorithm to reduce the cores' P-state in communication regions (*Tcomm*). *Fermata* uses a prediction algorithm to decide when scaling down the P-state; the prediction is determined by the amount of time spent in communication during the previous call. If the duration is greater than or equal to twice the switching threshold, *Fermata* sets a timeout to expire at the threshold time, instead if the task (MPI call) terminates before the timer expires, the callback is canceled. The threshold time is empirically set to 100ms. Calls are identified as specific MPI primitives in the application code through the hash of the pointer (*Task id*) that makes up the stack trace. The *Task id* is generated when the application encounters an MPI primitive; hence, each MPI primitive in the code is uniquely identified. The information about the last call is stored in a look-up table used to choose if to set the timer in the next call.

In *COUNTDOWN Slack* we implemented two versions of the Fermata policy, one with the original empirical switching threshold value of 100ms [16], and one with an empirical switching threshold tuned for the target system of 500μs [8].

5.2 Andante

Differently from *Fermata*, *Andante* [16] focuses on slowing down the computation region (*Tcomp*) to reduce the slack time (*Tslack*). This approach is based on the assumption that *Tcomp*, *Tslack*, and the number of instructions retired for a given task will be the same as the previous one for the same task. *Andante* computes the highest P-state for *Tslack* (aiming at reducing it) by exploiting the instructions per second (IPS) estimated based on measurements the previous time the same task was encountered (last-value prediction). *Andante* logic in [16] uses a pre-characterization of the message-transfer time of the MPI library to estimate the *Tcopy*. *Tslack* is calculated as the difference between *Tcomm* and *Tcopy*. Similarly to *Fermata*, *Andante* distinguishes tasks using the stack trace at the end of each collective MPI primitive. The information regarding the last executed task is kept in a look-up table containing the IPS for each discrete P-state of the system and the next P-state to assign.

COUNTDOWN Slack implements the same logic of *Andante*, but it uses the *Collective and P2P COUNTDOWN Slack barrier* to compute *Tslack* as the [16] pre-characterization step cannot be ported as it is in NUMA compute node.

5.3 Adagio

The idea behind *Adagio* [16] is to merge *Fermata* and *Andante* in a single energy-aware runtime. *Andante* slows down the computation regions, while *Fermata* handles the communication phases.

In *COUNTDOWN Slack* we implemented the same logic of *Adagio* by combining *Andante* and *Fermata*. We used in this case only *Fermata* configured with the empirical switching threshold at 500μs and applied only to the slack regions isolated with the *Collective and P2P Countdown Slack barrier* logic.

5.4 COUNTDOWN

COUNTDOWN [31] is a runtime library to identify and automatically reduce the power consumption of the computing elements during the communication phases. It uses

a timeout strategy to filter-out short communication regions (those too fast for the DFVS control knob to react, i.e., shorter than 500μs). *COUNTDOWN* differs from *COUNTDOWN Slack* as it considers the communication phase, while *COUNTDOWN Slack* focuses only on the slack time. However, the timeout implementation is similar.

6 EXPERIMENTAL RESULTS

6.1 Experimental Setup

6.1.1 Target Architecture

For all the experiments we use a Tier-1 HPC system based on an IBM NeXtScale cluster which is currently ranked in the Top500 supercomputer list [6]. The compute nodes of the HPC system, are equipped with 2 Intel Broadwell E5-2697 v4 CPUs, with 18 cores at 2.3 GHz nominal clock frequency and 145W TDP and 128 GB of DDR4. Each node runs the Centos 7 OS and Linux kernel 3.10.0, nodes are interconnected with an Intel QDR (40Gb/s) Infiniband high-performance network. We compile all our benchmarks using *GCC/GFortran 6.2* as our toolchain, coupled with *OpenMPI 3.2* as the communication library.

The default configuration for the power management in the target system is with the Linux *cpufreq* driver at the maximum P-state with turbo mode enabled. This is the baseline for our experimental results and we refer to this configuration lately as *Baseline*. It must be noted that this condition would be the same with any idleness-based power governors and MPI applications which busy waits during communication slack.

6.1.2 NAS Parallel Benchmark

The NAS Parallel Benchmark suite (NPB) is a set of popular HPC benchmarks developed by the NASA Advanced Supercomputing division. The NPB consist of 8 benchmarks and kernel namely BT, CG, FT, LU, SP, EP, MG and IS, which are widely used in different scientific areas such as spectral transform, fast Fourier transform, partial differential equations, fluid dynamics, and so on. We used NPB version 3.3.1 and tested different configurations to balance the duration of all benchmarks at around 10 minutes of execution time. For BT, CG, FT, LU, and SP we ran on 29 nodes using 1024 cores with data set E while for EP and MG we used the same data set but on four nodes and 128 cores. Instead, for IS we use data set D, which is the largest available one for that benchmark.

6.1.3 OMEN

OMEN is an atomistic quantum transport simulator that can compute the I-V characteristics of all kinds of nano-devices at the ab initio level (from first principles) [59], [60]. The code has been optimized to run on the largest available supercomputers, reaching two times the ACM Gordon Bell Prize final [33], [34]. Here, a transistor with a 2-D crystal as channel material serves as a benchmark.

We tested two configurations for OMEN, the above-mentioned called OMEN.1056p, which it runs on 29 nodes using 1056 cores and the OMEN.60p which runs on a couple of nodes with a scale-down data set.

Application	Tcomm [%]	Tslack [%]	*Fermata* 100ms [%]	*Fermata* 500µs [%]	CNTD [%]	CNTD Slack [%]	AVG MPI Time [ms]
nas_bt.E.1024	0.12	0.07	0.00	0.00	0.12	0.07	1.831
nas_cg.E.1024	34.84	0.07	0.39	32.68	32.96	0.01	2.068
nas_ep.E.128	7.56	7.56	0.00	0.00	7.56	7.56	24384.882
nas_ft.E.1024	65.10	12.28	55.88	57.80	65.09	12.28	2374.646
nas_is.D.128	62.73	27.42	31.14	40.98	62.65	27.41	277.003
nas_lu.E.1024	51.01	45.51	9.91	21.93	22.42	21.79	0.099
nas_mg.E.128	8.94	0.09	0.01	7.95	8.48	0.06	1.134
nas_sp.E.1024	0.05	0.02	0.00	0.00	0.05	0.02	1.447
omen_60p	59.69	56.00	43.87	48.86	59.60	55.99	59.853
omen_1056p	62.96	56.42	50.85	60.18	62.83	56.41	58.193

TABLE 2: Slack Isolation Potential and average MPI time.

6.2 Slack Isolation Policy Performance

As discussed in the previous subsection, it is difficult to obtain accurate predictions for *Tcomp*, *Tslack*, *Tcopy*, and *Tcomm*. The energy savings and performance neutrality depend on the capability of the given algorithm to (i) isolate the slack time from the copy time in MPI primitives, and (ii) avoid P-state transitions for a time shorter than $500\mu s$ [8], [31].

In this subsection, we analyze the capability of *COUNTDOWN Slack* in comparisons with state-of-the-art approaches in taking advantage of slack time to reduce the energy consumption and limiting the application overhead while discarding shorter slack regions and copy time regions. We conducted this analysis on the application traces recorded by *COUNTDOWN Slack Event Profiler* during the execution of the test applications with default node power management settings (*Baseline* configuration). These traces are the same as used in Subsection 3.4. On these traces we have implemented the *Fermata* and *COUNTDOWN*, as described in Section 5, as well as the *COUNTDOWN Slack* isolation and timeout policy. For the *Fermata* algorithm we report both versions with the empirical switching threshold set at 100ms (as described in [16]) and at $500\mu s$ (adapted to the characteristics of the target architecture [8], [31]).

Table 2 shows the results of this test. Each row corresponds to a different application, and the column reports the total *Tcomm* and *Tslack* time, as well as for each power management runtime analyzed the total time in which the algorithm is capable of reducing the power consumption. Values in each column are reported in percentage with respect to the execution time of the application. The column *AVG MPI Time* reports the average time duration of the MPI primitives in milliseconds.

From Table 2, we first notice that for the different applications (rows) the *Tslack* time is a sub set of the *Tcomm* time. For some applications (BT, GC, MG, SP, and FT of the NPB) the *Tslack* time is small fraction of the *Tcomm* timed, while for others (IS, LU and the OMEN benchmark) the *Tslack* time is significant.

As expected, we note that *Fermata* $500\mu s$ outperforms *Fermata* 100ms for all benchmarks as the 100ms empirical switching threshold was extracted by the authors of [16] on an older system with different power management characteristics than the one used in this study. We also highlight that in moving from 100ms to $500\mu s$ the potential energy saving increases drastically for the NPB, but less for the OMEN production runs. When comparing *Fermata* $500\mu s$ with *COUNTDOWN* we observe that the reactive timeout policy of *COUNTDOWN* is always more effective than the proactive timeout policy of *Fermata*, leading to remarkable additional energy savings up to 11% more for OMEN.60p

and 22% more for the is.D.128 case. It must be emphasized that both *Fermata* and *COUNTDOWN* are slack agnostic and thus cannot prevent the policy to slow down also *Tcopy* regions. Differently, the slack isolation policy proposed in *COUNTDOWN Slack* can separate the *Tslack* regions from the *Tcopy* ones. This is visible in Table 2 as *COUNTDOWN Slack* obtains in general lower coverage of the *Tcomm* and focuses only in the *Tslack*. We also note that for real application production run as OMEN.60p *COUNTDOWN Slack* is capable of capturing more power saving opportunities than *Fermata* $500\mu s$ even if *COUNTDOWN Slack* targets the slack time only. It is also interesting to underline that lu.E.1024, differently from the other applications is characterized by a large fraction of *Tcomm* ($> 50\%$), which is almost entirely *Tslack* (45%), but the application spends half of this time in *Tcomp* regions which are shorter than $500\mu s$ (visible from the column *AVG MPI Time*). This can be seen by *Fermata*, *COUNTDOWN*, and *COUNTDOWN Slack* which cannot exploit all the *Tcomm* time.

6.3 COUNTDOWN Slack Run-time Results

In this subsection we report the performance penalty (if any), power and energy saving of the proposed *COUNTDOWN Slack* power management runtime with respect to state-of-the-art approaches presented in Section 5 when applied to the different benchmarks.

We use as a baseline of our characterization the *Baseline* case, we also take into exam the case where all the nodes were configured to operate statically at the minimum available P-state (*Min Freq*). This is an important scenario as it allows to put in perspective the impact of policies that change the P-state in the computation regions, like *Adagio* and *Andante*. We report for each configuration the execution time overhead (Ex.Time Overhead), the average power saving, and the energy saving with respect to the *Baseline* case.

By looking at the *Min Freq* we notice that almost all benchmarks (excluded nas_ep.E.128, OMEN_60p, and OMEN_1056p) are memory bound as executing them at the minimum P-state always leads to an energy saving. Moreover, the *Min Freq* case shows, as expected, the highest power saving. This is not true for the energy saving as *Min Freq* causes, in general, a non-negligible overhead in the execution time.

If we first focus on the row of the table named average, which shows the average results for all the benchmarks, we can make the following observations:

(1) *Min Freq* induces the highest overheads in the application execution time. This is expected for *Min Freq* because the entire application is executed at the minimum P-state available.

(2) *Andante* and *Adagio* algorithms reduce the frequency on the *Tcomp* regions of the application, achieve the maximum power saving but lead to significant performance penalty, respectively in average 38.65% and 42.87%. This shows that the predictive logic of *Andante* is not capable of effectively estimating in advance the slack of the application regions and their instruction per second in today's real production HPC systems.

(3) The two approaches *Fermata* and *COUNTDOWN*, which are not aware of the *Tslack* but use a timeout based

Application	Ex.Time Overhead						Energy Saving						Average Power Saving					
	Min Freq	Fermata	Andante	Adagio	CNTD	CNTD Slack	Min Freq	Fermata	Andante	Adagio	CNTD	CNTD Slack	Min Freq	Fermata	Andante	Adagio	CNTD	CNTD Slack
nas_bt.E.1024	**72.18**	1.95	**77.72**	**68.94**	**8.92**	0.75	3.39	2.07	0.11	3.35	5.96	7.97	43.89	3.95	43.79	42.79	13.66	8.65
nas_cg.E.1024	**21.73**	3.86	**8.18**	**14.35**	4.23	1.08	21.59	18.89	24.72	22.69	22.58	9.57	35.59	21.91	30.41	32.39	25.72	10.54
nas_ep.E.128	**136.04**	-0.31	-0.15	1.30	0.80	-0.60	**-15.00**	0.62	0.10	**-1.35**	0.05	1.04	51.28	0.31	-0.05	-0.05	0.84	0.44
nas_ft.E.1024	**34.54**	2.57	**24.32**	**30.22**	3.50	0.26	20.89	23.59	18.25	17.76	25.92	6.25	41.20	25.51	34.24	36.85	28.42	6.50
nas_is.D.128	**29.95**	3.13	3.86	4.23	3.21	1.85	19.42	17.89	17.63	17.82	22.65	11.32	37.99	20.38	20.70	21.16	25.05	12.93
nas_lu.E.1024	**77.56**	**12.79**	**115.86**	**144.75**	**7.65**	3.02	3.82	**-9.96**	**-15.62**	**-24.69**	4.30	4.16	45.83	2.51	46.44	49.05	11.10	6.97
nas_mg.E.128	4.15	0.52	4.09	4.29	-0.14	0.03	22.58	6.41	7.83	13.71	10.68	1.57	25.82	7.09	11.64	17.43	10.74	1.81
nas_sp.E.1024	**12.44**	-0.07	**5.41**	**5.16**	-0.01	0.34	22.28	15.12	23.71	24.11	18.62	18.44	30.88	15.06	27.62	27.83	18.61	18.72
omen_60p	**120.65**	**5.01**	**108.65**	**114.44**	**8.81**	0.77	**-9.72**	15.12	**-20.19**	**-14.59**	17.33	17.14	50.27	19.18	42.40	46.56	24.03	17.77
omen_1056p	**42.12**	2.45	**38.59**	**41.04**	3.22	0.38	**-3.67**	20.99	**-2.09**	**-4.26**	24.72	22.11	0.71	26.63	0.99	1.33	34.28	22.92
AVG	**55.14**	3.19	**38.65**	**42.87**	4.02	0.79	8.56	11.07	5.45	5.46	15.28	9.96	36.35	14.25	25.82	27.53	19.24	10.73
WORST	**136.04**	**12.79**	**115.86**	**144.75**	**8.92**	3.02	**-15.00**	**-9.96**	**-20.19**	**-24.69**	0.05	1.04	0.71	0.31	-0.05	-0.05	0.84	0.44

TABLE 3: Comparison of execution overhead, energy, and average power saving using different approaches [%] with respect to the benchmark execution without power management strategy. We highlighted in bold and red Ex.Time Overhead not negligible ($> 5\%$) and energy losses.

policy, have a performance overhead of 3.19% and 4.02% respectively. While these two approaches have similar time to completion, the energy and power saving is lower on average and *COUNTDOWN* achieves an additional 4.21% of energy saving. *COUNTDOWN Slack* achieves a negligible performance overhead ($< 1\%$) with respect to *Fermata* and *COUNTDOWN* with a significant energy saving of 9.96%.

If we focus on the worst-case results, we can observe:

(1) *Min Freq* does not induce the highest overheads in the application execution time. This because *Andante* and *Adagio* algorithms can induce a non-negligible overhead caused by the hash of the call stack to generate the *Task id* in very short MPI communications (as instance in nas_lu.E.1024).

(2) The overhead of *Fermata* and *COUNTDOWN* approaches can be very significant, respectively 12.79% and 8.92%. It must be noted that *Fermata* is worse than *COUNTDOWN* for the time needed for computing the *Task id* used by its prediction algorithm. As effect of this *Fermata* induces an energy penalty of 9.96%. While *COUNTDOWN Slack* is always able to maintain a tolerable overhead for HPC applications, even in the worst case ($\leq 3\%$).

(3) *COUNTDOWN Slack* never induces energy penalty, while all the other approaches induce between 9.96% and 24.69% of energy penalty except *COUNTDOWN*.

If we now look at the individual benchmarks, we observe particular features that better describe the benefits of the proposed *COUNTDOWN Slack* algorithm with respect to the state-of-the-art approaches and *Baseline* case.

From the nas_lu.E.1024, we can see that the *Andante* algorithm induces a severe slowdown, which is even worse than the *Min Freq* case, this counter-intuitive result is originated by the overhead related to the task prediction algorithms, which requires to compute the *Task id*. This is critical in applications with a high density of MPI calls as shown in Table 2.

We conclude, as also suggested by the previous analysis of the predictability of the region duration, that proactive approaches are not suitable for performance-neutral energy-saving scenarios of supercomputers. On the contrary, the proposed algorithm *COUNTDOWN Slack* is effective in isolating the slack and filtering out short *Tcomp* regions leading to significant energy saving (up to 22.11% for large-scale production runs) with negligible overhead (always below 3%). This proves the effectiveness of the slack insertion logic combined with the timeout policy, making *COUNTDOWN Slack* performance neutral.

7 CONCLUSION

In this paper, we present *COUNTDOWN Slack*, a new power management runtime for scientific computing systems. *COUNTDOWN Slack* combines a novel slack insertion logic with a timeout policy for performance-neutral energy reduction in MPI-based applications. We tested *COUNTDOWN Slack* in a large set of HPC benchmarks extracted from the NAS parallel benchmark suite and with production runs of the two-times ACM Gordon Bell finalist, OMEN, a quantum-transport application. We compared the proposed approach with reactive and proactive power management libraries presented in the state of the art, showing that *COUNTDOWN Slack* can preserve the application execution time while reducing the energy consumed by the compute units on average by 9.96%. *COUNTDOWN Slack* allows discovering the communication slacks automatically, reducing the core's frequency, and saving energy. From our findings *COUNTDOWN Slack* is the only runtime that always leads to an energy saving (proportional to the communication slacks) with negligible execution time overheads ($< 3\%$).

ACKNOWLEDGMENTS

REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, May 2012.

[3] H. Shoukourian, T. Wilde, H. Huber, and A. Bode, "Analysis of the efficiency characteristics of the first high-temperature direct liquid cooled petascale supercomputer and its cooling infrastructure," *Journal of Parallel and Distributed Computing*, vol. 107, pp. 87 – 100, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517301272

[4] C. Conficoni, A. Bartolini, A. Tilli, C. Cavazzoni, and L. Benini, "Integrated energy-aware management of supercomputer hybrid cooling systems," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 4, pp. 1299–1311, Aug 2016.

[5] C. Conficoni, A. Bartolini, A. Tilli, C. Cavazzoni, and L. Benini, "Hpc cooling: A flexible modeling tool for effective design and management," *IEEE Transactions on Sustainable Computing*, pp. 1–1, 2018.

[6] J. J. Dongarra, H. W. Meuer, E. Strohmaier *et al.*, "Top500 supercomputer sites," [Online]; https://www.top500.org/lists, 2019, accessed 29 March 2019.

[7] W.-c. Feng and K. Cameron, "The green500 list: Encouraging sustainable supercomputing," vol. 40, no. 12.   IEEE, 2007.

[8] D. Hackenberg, R. Schne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the intel haswell processor," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 896–904.

[9] T. Rosedahl, M. Broyles, C. Lefurgy, B. Christensen, and W. Feng, "Power/performance controlling techniques in openpower," in *High Performance Computing*, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds.   Cham: Springer International Publishing, 2017, pp. 275–289.

[10] B. A. CESARINI, Daniele and L. BENINI, "Energy saving and thermal management opportunities in a workload-aware mpi runtime for a scientific hpc computing node," *Parallel Computing is Everywhere*, vol. 32, p. 277, 2018.

[11] "Advanced Configuration and Power Interface (ACPI) Specification," [Online]; http://www.acpi.info/spec.htm, 2019, accessed 29 March 2019.

[12] F. Fraternali, A. Bartolini, C. Cavazzoni, G. Tecchiolli, and L. Benini, "Quantifying the impact of variability on the energy efficiency for a next-generation ultra-green supercomputer," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ser. ISLPED '14.   New York, NY, USA: ACM, 2014, pp. 295–298.

[13] A. Auweter, A. Bode, M. Brehm, L. Brochard, N. Hammer, H. Huber, R. Panda, F. Thomas, and T. Wilde, "A case study of energy aware scheduling on supermuc," in *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds.   Cham: Springer International Publishing, 2014, pp. 394–409.

[14] C. Hsu and W. Feng, "A power-aware run-time system for high-performance computing," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Nov 2005, pp. 1–1.

[15] J. Eastep, S. Sylvester, C. Cantalupo, B. Geltz, F. Ardanaz, A. Al-Rawi, K. Livingston, F. Keceli, M. Maiterth, and S. Jana, "Global extensible open power manager: A vehicle for hpc community collaboration on co-designed energy management solutions," in *High Performance Computing*.   Springer International Publishing, 2017, pp. 394–412.

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

[17] D. Li, B. R. de Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos, "Hybrid mpi/openmp power-aware computing," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–12.

[18] F. Fraternali, A. Bartolini, C. Cavazzoni, G. Tecchiolli, and L. Benini, "Quantifying the impact of variability on the energy efficiency for a next-generation ultra-green supercomputer," in *International Symposium on Low Power Electronics and Design, ISLPED'14, La Jolla, CA, USA - August 11 - 13, 2014*, 2014, pp. 295–298. [Online]. Available: http://doi.acm.org/10.1145/2627369.2627659

[19] F. Fraternali, A. Bartolini, C. Cavazzoni, and L. Benini, "Quantifying the impact of variability and heterogeneity on the energy efficiency for a next-generation ultra-green supercomputer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1575–1588, 2018.

[20] A. Borghesi, A. Bartolini, M. Milano, and L. Benini, "Pricing schemes for energy-efficient hpc systems: Design and exploration," *The International Journal of High Performance Computing Applications*, vol. 0, no. 0, p. 1094342018814593, 0. [Online]. Available: https://doi.org/10.1177/1094342018814593

[21] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, "Scheduling-based power capping in high performance computing systems," *Sustainable Computing: Informatics and Systems*, vol. 19, pp. 1–13, 2018.

[22] N. Kappiah, V. W. Freeh, and D. K. Lowenthal, "Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Nov 2005, pp. 33–33.

[23] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in

[24] M. Maiterth, G. Koenig, K. Pedretti, S. Jana, N. Bates, A. Borghesi, D. Montoya, A. Bartolini, and M. Puzovic, "Energy and power aware job scheduling and resource management: Global surveyinitial analysis," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.   IEEE, 2018, pp. 685–693.

[25] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz, "Bounding energy consumption in large-scale mpi programs," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07.   New York, NY, USA: ACM, 2007, pp. 49:1–49:9.

[26] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, "Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Nov 2006, pp. 14–14.

[27] D. J. Kerbyson, A. Vishnu, and K. J. Barker, "Energy templates: Exploiting application information to save energy," in *2011 IEEE International Conference on Cluster Computing*.   IEEE, 2011, pp. 225–233.

[28] S. Bhalachandra, A. Porterfield, S. L. Olivier, and J. F. Prins, "An adaptive core-specific runtime for energy efficiency," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 947–956.

[29] A. Venkatesh, A. Vishnu, K. Hamidouche, N. Tallent, D. Panda, D. Kerbyson, and A. Hoisie, "A case for application-oblivious energy-efficient MPI runtime," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–12.

[30] D. Cesarini, A. Bartolini, P. Bonfà, C. Cavazzoni, and L. Benini, "Countdown: A run-time library for application-agnostic energy saving in mpi communication primitives," in *Proceedings of the 2Nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems*, ser. ANDARE '18.   New York, NY, USA: ACM, 2018, pp. 2:1–2:6.

[31] D. Cesarini, A. Bartolini, P. Bonfà, C. Cavazzoni, and L. Benini, "COUNTDOWN - three, two, one, low power! A run-time library for energy saving in MPI communication primitives," *CoRR*, vol. abs/1806.07258, 2018. [Online]. Available: http://arxiv.org/abs/1806.07258

[32] D. H. Bailey, *NAS Parallel Benchmarks*.   Boston, MA: Springer US, 2011, pp. 1254–1259. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_133

[33] M. Luisier, T. B. Boykin, G. Klimeck, and W. Fichtner, "Atomistic nanoelectronic device engineering with sustained performances up to 1.44 pflop/s," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11.   New York, NY, USA: ACM, 2011, pp. 2:1–2:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063387

[34] M. Calderara, S. Brck, A. Pedersen, M. H. Bani-Hashemian, J. VandeVondele, and M. Luisier, "Pushing back the limit of ab-initio quantum transport simulations on hybrid supercomputers," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–12.

[35] K. Hoga and B. Rountree, "Github scalability-llnl/msr-safe, 2014," [Online]; https://github.com/LLNL/msr-safe, 2019, accessed 29 March 2019.

[36] D. Hackenberg, T. Ilsche, J. Schuchart, R. Schne, W. E. Nagel, M. Simon, and Y. Georgiou, "Hdeem: High definition energy efficiency monitoring," in *2014 Energy Efficient Supercomputing Workshop*, Nov 2014, pp. 1–10.

[37] A. Libri, A. Bartolini, and L. Benini, "Dig: Enabling out-of-band scalable high-resolution monitoring for data-center analytics, automation and control (extended)," 2018.

[38] A. Borghesi, A. Libri, L. Benini, and A. Bartolini, "Online anomaly detection in hpc systems," in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, March 2019, pp. 229–233.

[39] M. Knobloch, M. Foszczynski, W. Homberg, D. Pleiter, and H. Böttiger, "Mapping fine-grained power measurements to hpc application runtime characteristics on ibm power7," *Computer Science - Research and Development*, vol. 29, no. 3, pp. 211–219, Aug 2014. [Online]. Available: https://doi.org/10.1007/s00450-013-0245-5

[40] A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier, "Powerapi: A software library to monitor the energy consumed at the process-level," *ERCIM News*, vol. 2013, no. 92, 2013.

[41] S. Mintchev and V. Getov, "Pmpi: High-level message passing in fortran77 and c," in *High-Performance Computing and Networking*, B. Hertzberger and P. Sloot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 601–614.

[42] V. Sundriyal, M. Sosonkina, and A. Gaenko, "Energy efficient communications in quantum chemistry applications," *Computer Science - Research and Development*, vol. 29, no. 2, pp. 149–158, May 2014.

[43] V. Sundriyal and M. Sosonkina, "Per-call energy saving strategies in all-to-all communications," in *European MPI Users' Group Meeting*. Springer, 2011, pp. 188–197.

[44] V. Sundriyal, M. Sosonkina, and Z. Zhang, "Achieving energy efficiency during collective communications," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 15, pp. 2140–2156, 2013.

[45] M. G. F. Dosanjh, R. E. Grant, P. G. Bridges, and R. Brightwell, "Re-evaluating network onload vs. offload for the many-core era," in *2015 IEEE International Conference on Cluster Computing*, Sep. 2015, pp. 342–350.

[46] J. Li, J. F. Martinez, and M. C. Huang, "The thrifty barrier: energy-aware synchronization in shared-memory multiprocessors," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, Feb 2004, pp. 14–23.

[47] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 299–316, June 2000.

[48] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, Mar 2014.

[49] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[50] J. Labarta, "New analysis techniques in the cepba-tools environment," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 125–143.

[51] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[53] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and regression trees. wadsworth & brooks," *Cole Statistics/Probability Series*, 1984.

[54] C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn, "Bias in random forest variable importance measures: Illustrations, sources and a solution," *BMC bioinformatics*, vol. 8, no. 1, p. 25, 2007.

[55] C. Strobl, A.-L. Boulesteix, T. Kneib, T. Augustin, and A. Zeileis, "Conditional variable importance for random forests," *BMC bioinformatics*, vol. 9, no. 1, p. 307, 2008.

[56] T. Parr, K. Turgutlu, C. Csiszar, and J. Howard, "Beware default random forest importances," 2018.

[57] S. Walker and M. McFadden, "Best practices for scalable power measurement and control," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1122–1131.

[58] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz, "Bounding energy consumption in large-scale mpi programs," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 49:1–49:9. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362688

[59] M. Luisier, A. Schenk, W. Fichtner, and G. Klimeck, "Atomistic simulation of nanowires in the $sp^3d^5s^*$ tight-binding formalism: From boundary conditions to strain calculations," *Phys. Rev. B*, vol. 74, p. 205323, Nov 2006. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.74.205323

[60] M. Luisier, "Atomistic simulation of transport phenomena in nanoelectronic devices," *Chem. Soc. Rev.*, vol. 43, pp. 4357–4367, 2014. [Online]. Available: http://dx.doi.org/10.1039/C4CS00084F

Daniele Cesarini received a Ph.D. degree in Electrical Engineering from the University of Bologna, Italy, in 2019. he is currently an HPC Software Engineer & Analyst in the HPC Business Unit of CINECA. His research interests concern the development of SW-HW codesign strategies as well as algorithms for parallel programming support for energy-efficient HPC systems.

Andrea Bartolini received a Ph.D. degree in Electrical Engineering from the University of Bologna, Italy, in 2011. He is currently Assistant Professor in the Department of Electrical, Electronic and Information Engineering (DEI) at the University of Bologna. Before, he was Post-Doctoral researcher in the Integrated Systems Laboratory at ETH Zurich. Since 2007 Dr. Bartolini has published more than 80 papers in peer-reviewed international journals and conferences with focus on dynamic resource management for embedded and HPC systems.

Andrea Borghesi is currently a Post-Doctoral researcher and Adjunct Professor at the Department of Computer Science and Engineering (DISI) of the University of Bologna, Italy. His research interests range broadly in the area of Artificial Intelligence, including Optimization, Power-awareness and Anomaly Detection in HPC systems, Scheduling and Allocation, Transprecision Computing, Machine and Deep Learning and Constraint Programming.
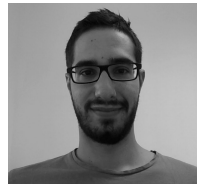
Carlo Cavazzoni graduated cum laude in Physics from the University of Modena and earned his PhD Material Science at the International School for Advanced Studies of Trieste in 1998. He has authored or co-authored several papers published in prestigious international review including Science, Physical Review Letters, Nature Materials. Currently in the HPC Business Unit of CINECA, he is responsible for the R&D, HPC infrastructure evolution and collaborations with scientific communities.

Mathieu Luisier is Associate Professor of Computational Nanoelectronics at ETH Zurich (Switzerland). His research focuses on the development of advanced technology computer aided design (TCAD) tools and their application to modern nano-devices such as next-generation transistors or non-volatile random access memory cells. He has published more than 200 articles in peer-reviewed journals and conferences and is a member of IEEE.

Luca Benini is professor of Digital Circuits and Systems at ETH Zurich, Switzerland, and a professor of electronics at at University of Bologna, Italy. His research interests are in the design of energy-efficient computing systems. He has published more than 1000 papers in peer reviewed international journals and conferences, five books and several book chapters. He is a fellow of the ACM and Member of the Academia Europea. He is the recipient of the 2016 IEEE CAS Mac Van Valkenburg Award.