Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: xxx/xxxx

REGULAR ARTICLE

On Power Capping and Performance Optimization of Multi-threaded Applications

Stefano Conoci[1] | Pierangelo Di Sanzo*[1] | Alessandro Pellegrini[1] | Bruno Ciciani[1] | Francesco Quaglia[2]

[1]DIAG, Sapienza, University of Rome, Italy
[2]DICII, University of Rome "Tor Vergata", Italy

Correspondence
*Corresponding author. Email: disanzo@diag.uniroma1.it

Summary

Multi-threaded applications facilitate the exploitation of the computing power of multicore architectures. On the other hand, these applications can become extremely energy-intensive, in contrast with the need for limiting the energy usage of computing systems. In this article, we explore the design of techniques enabling multi-threaded applications to maximize their performance under a power cap. We consider two control parameters: the number of cores used by the application, and the core power state. We target the design of an auto-tuning power-capping technique with minimal intrusiveness and high portability, which is agnostic about the workload profile of the application. We investigate two different approaches for building the strategy for selecting the best configuration of the parameters under control, namely a heuristic approach and a model-based approach. Through an extensive experimental study, we evaluate the effectiveness of the proposed technique considering two different selection strategies, and we compare them with existing solutions.

KEYWORDS:
Multi-threaded applications, power capping, performance optimization, energy efficiency, multicore architectures.

1 | INTRODUCTION

The ever-growing demand for computing services and applications has led to a relevant increase of the worldwide energy consumption by Information and Communication Technology (ICT) systems. A 2017 report[1] reveals that the latter has increased exponentially over the last years, and nowadays has reached approximately 7% of the global energy demand. In particular, data centers are estimated to have the fastest-growing carbon footprint from across the whole ICT sector, and the energy expenditure for their operations is estimated to have doubled in five years[2]. As a consequence, the need for limiting the energy used by computing systems has become a rising challenge. The objective of bounding the energy usage by a computing system can be achieved through the approach known as *power capping*. By forcing the power usage to stay below a given threshold, power capping enables to save energy and to keep costs under control.

Nowadays, most computing systems are based on multicore architectures. These architectures can effectively be exploited by multi-threaded workloads, which allow to take advantage of multi-core parallelism. The range of applications featuring multi-threaded workloads is wide, including, e.g., web applications, transactional applications, and HPC ones[3,4], for which performance is anyhow a key objective. Hence, one major challenge is to design effective power capping techniques suited for application contexts where parallelism exploitation and performance optimization are mandatory.

Modern multicore systems offer supports for controlling the power usage of the on-board cores. Examples include Dynamic Voltage and Frequency Scaling (DVFS), which allows to dynamically change the power state (in terms of voltage and frequency) of cores, and Clock Gating, which allows to disable portions of the CPU circuitry on-demand. Contextually, today's Operating Systems offer power management tools (like the Linux CPUFreq Governor[5]) allowing the direct control of DVFS. However, beyond the availability of these tools, it is not trivial to find the optimal configuration—in terms of number of cores simultaneously used and their power state—namely the one that provides the maximum performance of a multi-threaded application under power caps. This problem is further complicated by the fact that the workload profile of an application may change over time, leading in turn the optimal configuration to vary along the application execution. Accordingly, approaches aimed at identifying a static optimal setting may not be adequate.

In this article, we investigate the design of an auto-tuning technique for multi-threaded applications that dynamically optimizes the number of used cores and their power state under an arbitrary power cap. In our study we deal with workloads showing arbitrary scalability profiles, thus also including the complex scenario of limited-scalability applications. These are characterised by threads that conflict in the access to shared hardware resources (e.g. the same cache lines) and/or run computations that require to be synchronized (e.g. critical sections synchronized via locks or transactions). The key problem with limited-scalability applications is that increasing the number of used cores does not always lead to performance boost. Rather, it may lead to degraded performance, and consequently to poor energy efficiency because of the higher energy usage for powering the additional used cores.

The auto-tuning technique we designed does not need any off-line workload profiling phase, and requires to monitor at run-time only two parameters to make decisions: 1) the overall power usage by cores and 2) the specific performance metric of the application to be optimized (e.g. the throughput or the response time). This makes our technique easily usable and portable on common modern architectures. As for point 2, performance metrics that are not application-specific (such as CPU-level metrics like Instructions per Clock, which have been used in some literature solutions[6,7]), are not fully adequate for the optimization problem we target. In fact, many multi-threaded applications rely on mechanisms like spin-locking, non-blocking algorithms, or optimistic transaction execution. With these mechanisms, metrics like IPC can be tricked by busy waiting, or by transactions/operations abort and restart, which do not really perform useful application work.

We note that a trivial approach for finding the optimal configuration in terms of number of used cores and their power state is to explore the whole bi-dimensional configuration space of the two parameters. However, this might be prohibitive at run-time, especially when a large number of cores and/or power states are available. To address this issue, we designed and compared two different decision strategies, based on a heuristic approach and a model-based approach, respectively. The former is an exploration-based strategy that relies on some assumptions about the scalability profile of the workload to reduce the number of exploration steps. It has a linear time complexity $\mathcal{O}(\mathsf{p}_{\mathsf{max}} + \mathsf{n}_{\mathsf{max}})$, where $\mathsf{p}_{\mathsf{max}}$ is the number of available core's power states and $\mathsf{n}_{\mathsf{max}}$ is the number of available cores. We originally presented this exploration-based strategy in a previous study[8]. The second one is a model-based strategy, that we present for the first time in this article. It uses analytical models to predict the performance and the power usage for all the possible configurations, relying on measurements collected with a small subset of configurations. Its time complexity is $\theta(\mathsf{n}_{\mathsf{max}})$, thus it is independent of the number of power states. Also, differently from the exploration-based strategy, it does not rely on any assumption about the scalability profile of the workload. On the other hand, it has the potential disadvantage of being affected by model prediction errors.

In our study, we also observed that the effectiveness of a power-capping technique targeting the selection of the optimal configuration can be improved through an additional micro-tuning strategy. In fact, in our experiments we noted that, once selected a configuration, the power usage is typically subject to relatively small variations around its average value. If the latter is close to the target power cap value, then these variations likely lead to power cap violations. A micro-tuning strategy that, once the decision strategy has selected a configuration, can temporarily switch (if needed) to some adjacent configurations, can help limiting the violations without sacrificing performance. In this article, we presents a micro-tuning strategy which is completely decoupled from the selection strategy, thus working with both the exploration-based strategy and the model-based strategy.

To evaluate the effectiveness of the power capping technique with the different strategies we designed, we present the results of an extensive experimental study with various multi-threaded application benchmarks—spanning from highly-scalable workloads to workloads with extremely limited scalability—and we also make a comparison with other power capping techniques proposed in literature.

The remainder of this article is structured as follows. In Section 2, we analyse literature proposals. In Section 3, we provide some preliminaries notions to introduce our power capping technique. In Section 4, we illustrate the power capping technique, the two decision strategies (heuristic and model-based) and the micro-tuning strategy. The experimental evaluation is presented in Section 5.

2 | LITERATURE OVERVIEW

Some literature techniques, as well as various tools currently in use in commercial off-the-shelf processors, aim at limiting the power usage in an application-agnostic manner[9,10]. They regulate the core power state or disable unused components (e.g. turning off an entire core package) to keep the power usage below the target value. Consequently, they do not represent suitable solutions to optimize the application performance under a power cap. On the opposite side, some tuning techniques for performance optimization of multi-threaded workloads have been proposed for specific categories of applications and systems [11,12,13,14]. However, they only target performance objectives, assuming that no power constraints exists.

A technique proposed to optimize the performance of multi-threaded applications under a power cap is called Pack and Cap [15]. It controls at run-time the number of used cores and their power state. The authors of this technique show in their study that the configuration offering the highest application throughput under the power cap is the one that uses the highest number of available cores. However, the limitation of the presented study is that it does not cope with limited-scalability workloads, for which, as we show in this article, using the highest number of available cores does not always pay off. We used Pack and Cap in our comparative experimental study, and we show its drawbacks compared to our proposal.

The work presented by Zhang et al. [16] considers the problem of maximizing performance under a power cap while also taking into account the workload scalability profile. This solution defines an ordered set of configuration parameters that are tuned one after another, each one via binary search. This approach may fail at identifying the configuration ensuring the maximum performance just because the configuration parameters are tuned independently of each other. In our experimental study, we show the drawbacks of this tuning scheme compared to our technique.

Portfield et al. [17] present a technique that reduces the energy usage of OpenMP programs by throttling threads when both power and memory bandwidth usage is high. Throttled threads are put in a low-power mode by modifying the duty cycle of individual cores. This is a lightweight approach that, however, achieves low power reduction compared to pausing threads or modifying the frequency/voltage of cores. Furthermore, the proposed technique does not tune the power state of the cores running non-throttled threads and does not generally allow to meet a power cap.

Some literature contributions rely on Intel RAPL [18,9] as a building block to enforce power capping at hardware level for various subsystems (e.g. CPU package or memory). They estimate the power usage by observing different low-level hardware events, and then select the optimal power state of cores such that the average power usage of a specific subsystem is lower than its associated power cap. We decided not to rely on the power capping capabilities offered by RAPL, since they operate globally on the whole set of cores in the machine. Rather, as we already mentioned, we explicitly target the selection of the most suited number of cores to be used depending on the application scalability profile. Hence, we need to control the power state of arbitrary subsets of cores in the machine to meet the power cap. Also, RAPL is a proprietary technology only supported by recent Intel x86 processors. Conversely, we rely on the abstraction of *P-state*, which is a standard supported by processors of different manufacturers and with different instruction sets. Actually, in our experimental study we use RAPL exclusively as a power measuring tool on x86 processors.

Gholkar et al. [19] propose a 2-level hierarchical technique based on RAPL that uses an exploration-based approach to optimize the performance of a cluster under power constraints. This technique partitions the power budget of the cluster across different jobs. Then, for each job, it determines the set of nodes for assigning the job, and configures the node power level via RAPL. Differently from our proposal, this technique operates at cluster-level and does not deal with optimizing parallelism level within a single multi-core machine based on a target application performance metric. The work by Bari et al. [20] proposes an exploration-based technique that optimizes performance under a power cap for OpenMP applications. For each parallel region, it selects the appropriate number of threads, the scheduling policy and the chunk size using the Nelder-Mead search algorithm [21]. The search is performed at a fixed power-cap setting. Given that the search space may be large, it searches within a restricted space (e.g. 2, 4, 8 or 16 threads), which is established in advance to reduce the computation time. Differently from our proposal, this approach does not ensure to find the optimal setting.

Other works investigate the problem of improving the application performance under power constraints considering different power management variables. FastCap [7] defines an approach for optimizing performance under a system-wide power cap considering both CPU and memory DVFS. It defines a non-linear optimization problem solved through a queuing model that takes into account the interaction between cores and memory banks communicating over a shared bus. Unfortunately, although memory DVFS has not been proposed recently [22,23], it is not currently available in commercial systems. This limits the applicability of the proposed solution.

Kanduri et al. propose the usage of approximation in computation as another trick to provide suited trade-offs between performance/power-usage and accuracy of the results [24]. Obviously, this approach is applicable only when approximated results are acceptable.

PPEP [6] is an online prediction framework that, based on hardware performance events and on-chip temperature measurements, estimates the CPU performance and power usage for each different *P-state*. It allows implementing power capping techniques that can meet power targets in a single step without requiring any exploration. However, it operates with the number of core assigned by the Operating System to the application, thus it does not embed capabilities for explicitly tuning the number of cores to be used by an application along its execution.

Finally, the work by Li et al. [25] provides a model for estimating the effects on performance by different settings of the CPU and memory frequency. This work is tailored to performance prediction of bulk synchronous parallel applications and does not consider the problem of matching a power cap at runtime.

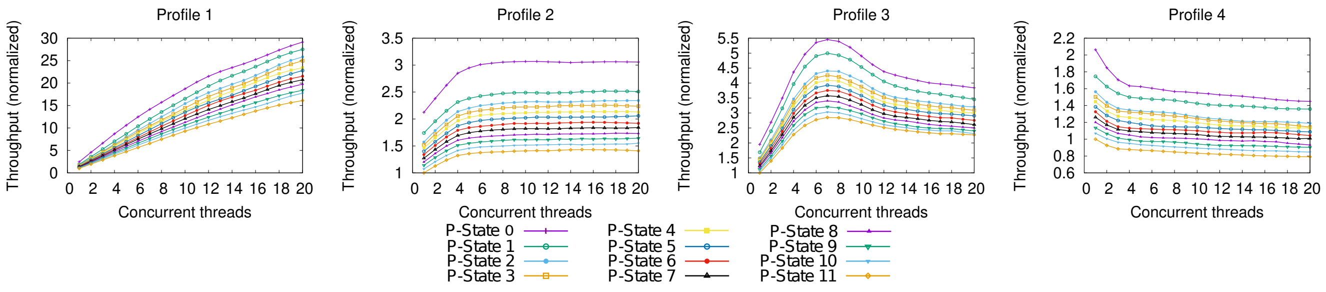

Figure 1 Throughput vs. Number of Concurrent Threads

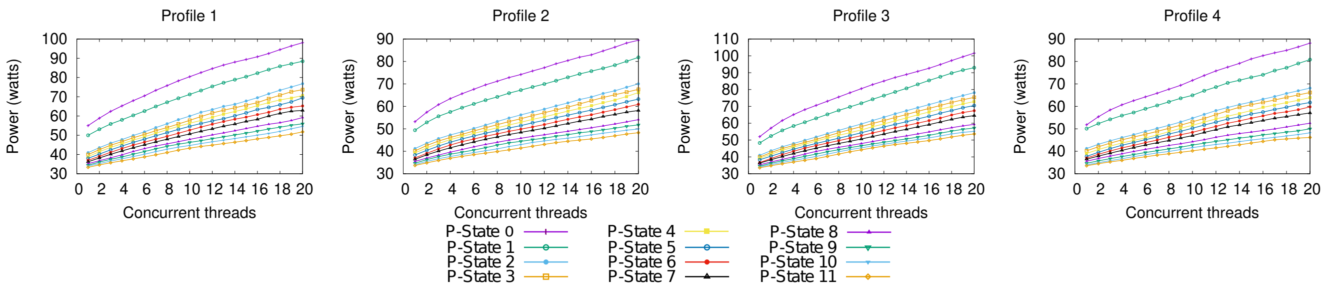

Figure 2 Power usage vs. Number of Concurrent Threads

3 | PRELIMINARIES

We adopt the ACPI Standard notation[26] for power states (P-states) of cores controlled via DVFS. Accordingly, we use *P0* to identify the P-state with maximum power usage and operating frequency and *P1*, *P2*, ... to progressively identify the P-states with less power usage and operating frequency. We also remark that in modern processors, when cores are unused, they transit from the full operating mode (the C-State known as C0) to some deeper sleep mode (the C-States known as C1, C2, ...), in which the core power demand is highly reduced. This may happen when the application runs with less threads than the number of cores available in the machine, or when the application threads are pinned to a subset of the available cores.

In order to build a knowledge-base to develop our power capping technique, we observed for various multi-threaded benchmark applications how P-state variations, in combination with the variation of the number of used cores, affect the throughput curves and the power usage. We have run the benchmark applications on a multi-core machine with two Intel Xeon E5-2630 v4, 20 physical cores total, with core frequency ranging from 1.2 GHz (the corresponding P-state is denoted as P11) to 2.2 GHz (the corresponding P-state is denoted as P1), Intel TurboBoost ranging from 2.2 GHz to 3.1 GHz (the corresponding P-state is denoted as P0), 256GB of DDR4 memory, running Debian 10.2 with Linux Kernel 4.19.0.

In Figure 1, we report the main profiles of the throughput curves that we observed with applications taken from the NAS Parallel benchmark[27] and the STAMP benchmark[28], which are representative of differentiated scalability profiles of multi-threaded applications. As for the applications of STAMP benchmark, we considered both their implementation based on Transactional Memory (TM)[1], as well as the less scalable implementation that uses a coarse-grained locking scheme in place of transactions. To draw the profile curves, we executed 10 runs of each applications with a fixed P-state and number of used cores, then we calculated the average throughput and the average power usage for each configurations. The plots show the normalized throughput curves, for different P-states, while increasing the number of concurrent threads. The normalized throughput for a configuration c is calculated as the ratio between the throughput measured with configuration c and the throughput measured with the configuration with 1 thread and the P-state with the lowest power usage and frequency, i.e. P11.

Profile 1 is representative of highly scalable applications, in which the throughput increases up to the number of available cores. *Profile 2* is representative of applications with limited-scalability where, exceeding a certain number of used cores, the throughput does not grow further. *Profile 3* is representative of applications with limited-scalability, for which the throughput decreases after reaching a maximum value. Finally, *Profile 4* is representative of applications with a workload that does not scale at all, i.e. the throughput decreases even if only two cores are used. A

[1]The TM implementation we used is based on the TinySTM package[29]

relevant aspect for all the profiles is that the P-state variation appears not to change the shape of the throughput curve. In particular, the number of concurrent threads for which the throughput curves reach the maximum values does not change when changing P-state. The corresponding curves depicting the power usage are shown in Figure 2. As expected, they show that the power usage always increases as a function of the number of concurrent threads or when decreasing P-state. This behavior is independent of the scalability profile. The sample applications from which we derived the representative profiles of Figure 1 and Figure 2 are Block tridiagonal solver (NAS benchmark) for *Profile 1*, the TM-based implementation of Genome (STAMP benchmark) for *Profile 2*, the TM-based implementation of Intruder (STAMP benchmark) for *Profile 3* and the lock-based implementation of Vacation (STAMP benchmark) for *Profile 4*.

As we will show, the configuration selection strategy based on the heuristic approach relying on exploration will result highly effective in all the scenarios where the application exhibits a behavior falling into whichever of the four shown scalability profiles. However, it may be possible to find applications whose behavior (also depending on the settings of the underlying hardware) does not fall into any of the analyzed profiles since their throughput curves show local maxima (i.e. they are not unimodal, see e.g.[30]), even though we anyhow expect their power usage to increase while increasing the number of used cores and/or while decreasing P-state. Our configuration selection technique is still able to cope with these scenarios when exploited in combination with the model-based strategy. Overall, what we present in this article is a folding fan of solutions able to address performance optimization under a power cap in generic application contexts where assumptions on the shape of the throughput curve may either hold or not.

4 | THE POWER CAPPING TECHNIQUE

We denote a configuration as (p, n), where p is a P-state and n is the number of used cores. $\mathrm{pwr}(p, n)$ denotes the power usage of configuration (p, n). Given a power cap value C, and assuming that S is the set of all possible configurations, we denote as $S_{ac} \subseteq S$ the subset including the acceptable configurations, i.e. the configurations for which the power cap is not violated. Hence, $\mathrm{pwr}(p, n) \leq C$ for each $(p, n) \in S_{ac}$.

The goal of our power capping technique is to dynamically select the configuration in S_{ac} that maximizes the application performance over time. Without loss of generality, we select the application throughput as target performance metric. Other performance metrics (e.g. response time) can anyhow be used with our technique. We denote as $\mathrm{thr}(p, n)$ the application throughput when running with configuration (p, n). Our technique periodically runs a selection procedure, that implements the selection strategy. The procedure returns a configuration $(p, n)^* \in S_{ac}$ that is expected to be the optimal one based on the current workload profile of the application. The period to re-run the selection procedure can be configured by the user. A good approach is to set at run-time this period equal to a given number N of times the duration of the selection procedure. This way, the overhead due to the selection procedure can be limited, along a time period, according, by the user-selected factor 1/N. In Section 5, we will enter more in details of this aspect.

As already mentioned, we devise two alternative selection strategies, both presented in the following sections.

4.1 | Exploration-based Strategy

The exploration-based strategy is an extension of the popular auto-tuning approach based on the hill-climbing search[31]. This search method has been used in various literature proposals and in existing systems to optimize a variety of parameters, including the number of concurrent threads[32,33] (as in Microsoft .NET Thread Pool[12]). In its basic version, the hill-climbing search repeatedly performs the following steps: 1) starting from the current configuration, it moves to a neighbour configuration, and 2) if the new configuration is better than the previous one (according to the metric to be maximized) it persists in the new configuration, otherwise it tries with another neighbour configuration. It terminates when none of the neighbour configurations provides advantages. With the basic hill-climbing version, it is assumed that the shape of the function to be maximized is unimodal. Based on this assumption, no local maxima exist, thus the selected configuration corresponds to the global maximum.

To cope with local maxima, a number of alternative versions have been designed, typically based on stochastic mechanisms. Examples include Simulated Annealing[34], Tabu Search[35], Iterated Local Search[36]. However, these methods only contribute to increase the probability to find the global maximum, and typically require much higher computation time to converge. This discourages their adoption in online self-tuning techniques. Conversely, various studies have shown that the basic hill-climbing version is effective in many cases.

Our exploration-based strategy is built on top of a search scheme similar to the basic hill-climbing. However, we have extended it to cope with the more complex optimization problem we target, where the configuration space is bi-dimensional and there is a constraint, namely the power cap. In addition to the unimodality of the speed-up curve, our strategy assumes that the power usage always increases as a function of the number of concurrent threads or when decreasing P-state, which is compliant to what shown in Figure 2. Moreover, it exploits our experimental findings on the shape of the throughput function when changing the number of concurrent thread, which we found to generally not change for different CPU P-states.

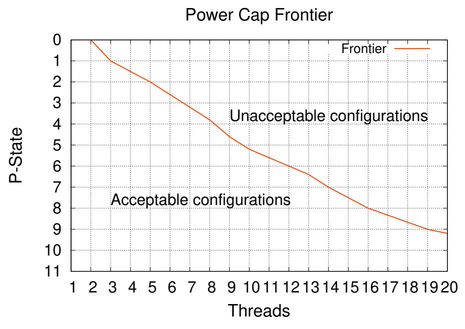

Figure 3 Example of a frontier line dividing acceptable and unacceptable configurations.

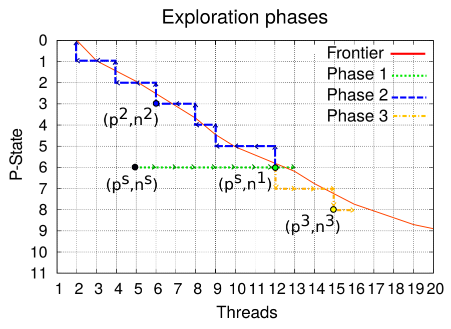

Figure 4 Example of exploration phases performed by the exploration strategy.

By relying on the above assumptions, the exploration-based strategy is able to find the optimal configuration by exploring only a subset of the whole bi-dimensional configuration space (the Supplemental Material section of this article includes a formal proof of the optimality of the selected configuration). In short, it moves along a well suited path of configurations and records the measures of the power usage and the throughput of the application. At the end of the exploration, the explored configuration with the highest throughput that does not violated the power cap is selected.

We now enter the details of the selection strategy. By the above-mentioned assumptions about power usage, $\mathsf{pwr}(p, n)$ is a monotonically increasing function of n and a monotonically decreasing function of p. Thus, the subsets of acceptable and unacceptable configurations are separated by a frontier line. In Figure 3, we show a frontier line example.

To identify the configuration $(p, n)^*$, the exploration-based strategy takes as input a starting configuration (p^s, n^s). Given that the strategy can be run periodically to re-adapt the selected configuration to the workload current profile, in the first execution the starting configuration can be selected randomly, while in subsequent executions it can coincide with the last selected configuration.

Exploiting the assumptions on the unimodality of the speed-up curves and on the increment of the power usage curves, some configurations can be safely pruned during the exploration according to the following rules:

1. if a configuration (p', n') such that $\mathsf{thr}(p', n') \leq \mathsf{thr}(p', n' - 1)$ is found, then all the configurations (p, n) where $n \geq n'$, for whichever p, can be pruned—because we are in the descending part of the throughput curve and the throughput curves preserve the shape while varying P-state;

2. if a configuration (p', n') such that $\mathsf{pwr}(p', n') \leq C$ is found then all configurations (p, n') with $p > p'$ can be pruned—because increasing P-state reduces the application throughput;

3. if a configuration (p', n') such that $\mathsf{pwr}(p', n') > C$ is found then all configurations (p, n) where $n \geq n'$ and $p \leq p'$ can be pruned—because decreasing P-state or increasing the number of used cores increases the power usage.

The exploration-based strategy works in 3 phases plus a final selection step, all of them described below. A graphical example to help the reader is shown in Figure 4, which refers to a test case where each throughput curve (associated to a given P-State) reaches the maximum value with 15 threads.

Phase 1. This phase starts from configuration (p^s, n^s) and returns a configuration that we denote as (p^s, n^1). It keeps p^s fixed and explores along the dimension of the number of cores. Specifically, starting from (p^s, n^s), it increments the number of cores while the throughput increases and the power cap is not violated. We note that if after an increment of the number of cores the throughput increases, this means that the search is moving along the ascending part of the throughput curve. This phase stops as soon as the throughput decreases, or the power cap is violated, or the maximum number of cores has been reached. Thus, it selects and returns the explored configuration for which it measured the highest throughput, which does not violate the power cap. On the other hand, if the throughput decreases after the first increment along the number of cores axis, or the power cap is violated, it changes the exploration direction and starts reducing the number of cores. Then, it returns the explored configuration with the highest throughput that does not violate the power cap. Otherwise, if all the explored configurations violate the power cap, or if the search reaches a number of cores equal to 1, it returns $(p^s, 1)$. In the example in Figure 4, phase-1 exploration is represented by the green line. It starts with $(p^s, n^s) = (6, 5)$, then increases the number of cores and terminates when it reaches configuration $(6, 13)$ since it violates the power cap. It returns $(p^s, n^1) = (6, 12)$, which satisfies the power cap.
Phase 2. This phase starts exploring from the configuration (p^s, n^1) returned by phase 1. Phase 2 is executed only if (p^s, n^1) does not violate the power cap (otherwise we jump to the next phase). The goal of phase 2 is to continue the exploration along lower values of P-state (we remark that reducing P-state leads to higher core performance and higher power usage). Specifically, we move from the current configuration (p^s, n^1) to configuration $(p^s - 1, n^1)$. If the latter configuration does not violate the power cap, we continue reducing the value of P-state. If a configuration that violates the power cap is reached, we start reducing the number of cores (in order to reduce the power usage) until the power usage falls below the power cap. Then, we restart the exploration again by decreasing P-state. The exploration in phase 2 terminates when p reaches 0 and the current configuration does not violate the power cap, or when configuration $(0, 1)$ is reached, or when some configuration with $n = 1$ violates the power cap. Then, among the explored configurations, phase 2 returns the configuration with the highest throughput that does not violate the power cap, or $(0, 1)$ if all the explored configurations violate the power cap. We denote the returned configuration as (p^2, n^2). In Figure 4, phase-2 exploration is shown by the blue line. It starts from $(p^s, n^1) = (6, 12)$, and then explores up to configuration $(0, 1)$. It returns $(p^2, n^2) = (3, 6)$.
Phase 3. This phase starts again from the configuration returned by phase 1, i.e. (p^s, n^1), and explores along higher values of P-state. This phase is executed only if phase 1 has explored along the ascending part of the throughput curve—namely, the throughput did not decrease or the maximum number of cores has been reached. In phase 3 we increment P-state by one and start increasing the number of cores until the power cap is violated or the throughput decreases. In the former case, if the maximum value of P-state has not been reached, we increment P-state by one and start again increasing the number of cores. In all the other cases the exploration terminates. Then, phase 3 returns the explored configuration with the highest throughput that does not violates the power cap, or it returns (p_{max}, n^1) if all the explored configurations are within the power cap. We denote the returned configuration as (p^3, n^3)). In Figure 4, phase-3 exploration is represented by the yellow line. It starts from $(p^s, n^1) = (6, 12)$, then moves up to configuration $(8, 16)$, where it stops since the throughput decreases (we remark that in the example the number of used cores providing the highest throughput is equal to 15). It returns $(p^3, n^3) = (8, 15)$.
Final phase. This phase simply selects the configuration with the highest throughput among the configurations (p^s, n^1), (p^2, n^2) and (p^3, n^3), which does not violate the power cap, or returns *null* if all of them violate the power cap. The scenario of no configuration satisfying the power cap (e.g. because of unfeasible power cap specification) clearly needs to be resolved by user defined mechanisms that are out of the scope of our work.

4.1.1 | Time Complexity

In this section, we estimate the time complexity of the exploration-based strategy in terms of the number of exploration steps required to find the optimal configuration. This can be done by estimating the time complexity of each exploration phase separately.

- **Phase 1.** Each configuration with a different number of used cores and $p = p^s$ is explored at most once, thus the time complexity is $\mathcal{O}(n_{max})$;
- **Phase 2.** Starting from a configuration (p, n), phase-2 exploration either reduces the value of p or reduces n. Starting from the configuration returned by phase 1, it can reduce p at most p_{max} times, and can reduce n at most n_{max} times. Thus, the time complexity of phase 2 is $\mathcal{O}(p_{max} + n_{max})$;
- **Phase 3.** Starting from a configuration (p, n), phase-3 exploration either increments the value of p or increments n. Thus, according to the same reasoning used for phase 2, the time complexity of phase 3 is $\mathcal{O}(p_{max} + n_{max})$.

In any case, the total number of exploration steps executed by phase 2 and phase 3 is at most $n_{max} + p_{max}$, thus the maximum number of steps, including phase 1, is $p_{max} + 2n_{max}$ Hence, the time complexity of the exploration-based strategy is $\mathcal{O}(p_{max} + n_{max})$.

4.2 | The Model-based Strategy

The model-based strategy relies on two analytical models used to estimate throughput and power usage. The strategy works in 3 phases. In the first phase, throughput and power usage are measured for a predetermined subset of configurations. In the second phase, the two models are instantiated by solving a system of equations and using the set of collected measures. In the third phase, throughput and power usage for all the configurations are estimated through the two analytical models, then the configuration estimated to provide the highest throughput and not violating the power cap is selected. The model-based strategy does not require any of the assumptions of the exploration-based strategy. Furthermore, the number of configurations for which throughput and power usage measurements have to be collected is linear with respect to the number of cores.

4.2.1 | Performance model construction

Our performance model allows the prediction of the throughput of a given execution phase of the application for whichever configuration. As a first step, the model estimates the CPU-bound portion and the memory-bound portion of the current workload based on measurements collected while running with two different arbitrary P-states. Once estimated these portions, it can predict the throughput for all the other P-states. At run-time, the model can be periodically re-instantiated to capture possible variations of the application workload that modify the ratio between the CPU-bound portion and the memory-bound portion. In the following, we enter into the details of the construction of the model.

In literature, various performance models for DVFS have been proposed. A classification is presented in[37]. Validation studies show that it is reasonable to assume a linear relation between the core frequency and the execution speed-up of the CPU-bound portion of the application.

Hence, the execution time T of a computation task running on a single core can be estimated as the sum of two parts, i.e.

$$T = C + M \tag{1}$$

In the above equation, C is the execution time of the CPU-bound portion of the task operations. Thus, C is subject to the variation of the operation execution speed when changing the core frequency. M is the execution time of all operations that are not affected by frequency variations of the core, such as memory operations. Thus, if we measure C for P-state p^j, denoting with f_j the associated core frequency, the execution time T_k of the computation for another P-state p^k with frequency f_k can be estimated as

$$T_k = \frac{f_j}{f_k} \cdot C + M \tag{2}$$

By Equation 2, once estimated C and M for an arbitrary P-state, we can estimate the computation execution time for whichever P-state. However, we note that, fixed P-state, C and M can vary when the number of cores that are running computation tasks in parallel changes. This is essentially due to the different level of contention on shared resources (e.g. shared caches and memory interconnection).

For this reason both C and M have to be estimated for each possible number of available cores. Thus, Equation 2 has to be rewritten as

$$T_{k,n} = \frac{f_j}{f_k} \cdot C_n + M_n \tag{3}$$

where $T_{k,n}$ is the execution time for P-state k when running with n cores, and C_n and M_n are the values of C and M when running with n cores, respectively.

In order to use the above model in our model-based configuration selection strategy, we have to rewrite it in terms of throughput rather than execution time. Since the throughput can be expressed as the number of computation tasks per time unit, it can be calculated as $1/T_{k,n}$.

Hence, by Equation 3, the throughput for a generic P-state p^k when running with n cores can be expressed as

$$thr(p^k, n) = \frac{1}{\frac{f_j}{f_k} \cdot C_n + M_n} \tag{4}$$

Considering that the throughput for P-state p^j is

$$thr(p^j, n) = \frac{1}{\frac{f_j}{f_j} \cdot C_n + M_n} = \frac{1}{C_n + M_n} \tag{5}$$

by solving the system composed of Equation 4 and Equation 5, we can calculate C_n and M_n as

$$C_n = \frac{f_{p^k} \cdot (thr(p^j, n) - thr(p^k, n))}{thr(p^j, n) \cdot thr(p^k, n) \cdot (f_j - f_k)} \tag{6}$$

$$M_n = \frac{1}{thr(p^j, n)} - C_n \tag{7}$$

Hence, we can calculate C_n and M_n for each n by measuring the throughput for two different arbitrary P-states p^k and p^j, and, for each of them, for each number of cores from 1 to n_{max}. Once calculated C_n and M_n, we can calculate the throughput $thr(p^l, n)$ for whichever configuration (p^l, n) with core frequency f_l using Equation 4, i.e.

$$thr(p^l, n) = \frac{1}{\frac{f_j}{f_j} \cdot C_n + M_n} \tag{8}$$

In practice, once taken the above-mentioned measurements, the model is able to predict the throughput curves for whichever P-state. Essentially, this is achieved by projecting the measured curves on the basis of the specific frequency of the P-state and the estimated values of C_n and M_n. This allows the model to predict the throughput curves independently of their specific profile, thus making the model able to capture whichever of the profiles shown in Figure 1.

4.2.2 | Power-usage model construction

The power-usage model is designed to allow the estimation of the power usage for a given execution phase of the application for each P-state. To be instantiated, it requires the measurement of the power usage with just two different arbitrary P-states, then it is able to estimate the power usage for all the other P-states. Thus, the model can be instantiated in real-time by taking at runt-time the required measurements. Similarly to the performance model, the power-usage model can be periodically re-instantiated to capture possible variations of the application workload that may lead to variations of the power usage for the different P-states. In the follow, we discuss the construction of the model.

A power-usage model for DVFS should be able to capture the effects on power usage due to CPU frequency variation. Literature studies (e.g.[38,39]) show that the power usage of a computer system can be approximated by the sum of three components. The first one is associated with the CPU dynamic power usage, which is due to switching of CPU transistors during the operation executions. It can be approximated as

$$P_{dyn} = \rho \cdot C_L \cdot V^2 \cdot f \tag{9}$$

where ρ is the switching probability, C_L is the system capacitance, V is the supply voltage and f is the frequency. Since a linear relation exists between the frequency f and the supply voltage V, by the above equation a cubic dependence exists between dynamic power usage and frequency. Accordingly, the power usage can be modeled as

$$P_{dyn} = \alpha \cdot f^3 \tag{10}$$

where α is a factor that depends on the specific hardware design and the workload profile. The second component is associated with the CPU leakage power, and can be approximated as $P_{leak} = V \cdot N \cdot k_{design} \cdot I_{leak}$, where N is the number of transistors, k_{design} is a CPU design-dependent parameter, I_{leak} is a technology-dependent parameter. By the linear dependence between V and f, this component can be modeled as

$$P_{leak} = \beta \cdot f. \tag{11}$$

Also in this case, β is a factor that can be assumed to be dependent on the hardware design and the workload profile. Finally, the thirty component represents the power usage of the system that is independent of frequency variations. To our aim, it can be considered as a fixed component, that we denote as P_{fixed}. Hence, the total power usage is calculated as

$$P = \alpha \cdot f^3 + \beta \cdot f + P_{fixed} \tag{12}$$

To use the above model in our model-based selection strategy, we need to estimate α, β and P_{fixed}. The latter can be estimated by measuring the power usage when all cores are idle. As for α and β, we use the same approach as for the performance model. By our experiments, we noted that also α and β vary depending the number of used cores. Thus, we estimate them as a function of the number of cores, using the notation α_t and β_t, respectively.

Fixed the number of cores n, we consider the power usage for two different P-state values p^j and p^k, respectively, and we subtract P_{fixed} from both of them. We denote the results as P^j_p and P^k_p. Thus, we build a system with the following equations, derived from Equation 12 after having subtracted P_{fixed} from the left term:

$$P_{p^j} = \alpha_t \cdot f^3_{p^j} + \beta_t \cdot f_{p^j} \tag{13}$$

$$P_{p^k} = \alpha_t \cdot f^3_{p^k} + \beta_t \cdot f_{p^k} \tag{14}$$

Solving the system we have

$$\alpha_t = \frac{P^j_p \cdot f_{p^k} - P^k_p \cdot f_{p^j}}{f_{p^k} \cdot {f_{p^j}}^3 - {f_{p^k}}^3 \cdot f_{p^j}}, \beta_t = \frac{P^j_p \cdot {f_{p^k}}^3 - P^k_p \cdot {f_{p^j}}^3}{{f_{p^k}}^3 \cdot f_{p^j} - f_{p^k} \cdot {f_{p^j}}^3}. \tag{15}$$

Once calculated α_t and β_t for any number of cores, we can estimate the power usage for each P-state p^l by calculating $P_{p^l} = \alpha_t \cdot f^3_{p^l} + \beta_t \cdot f_{p^l}$ and then adding P_{fixed}.

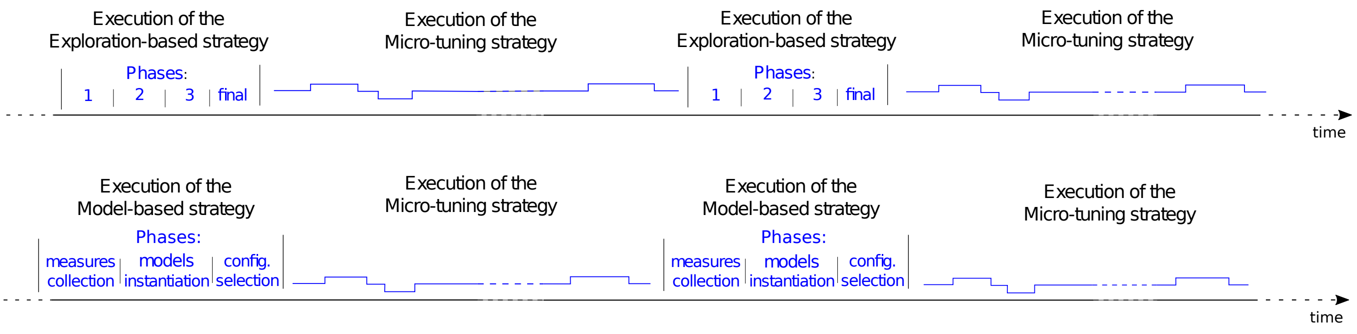

Figure 5 Example of the execution timelines of the technique with the exploration based strategy (top timeline) and the model based strategy (bottom timeline)

Finally, we note that the power-usage model can be instantiated using power measurements taken for the same set of configurations used to instantiate the performance model.

4.2.3 | Time Complexity

We observed that the time to solve the models to calculate the expected throughput and power consumption for all the configurations is negligible with respect to the time to instantiate them. Thus, we estimate the time complexity in terms of number configurations for which throughput and power usage have to be measured to instantiate the models. To this aim, given the models' construction, $2 \cdot \mathrm{n_{max}}$ configurations are required. Thus, $2 \cdot \mathrm{n_{max}}$ measurement steps have to be executed, and consequently the time complexity is $\theta(\mathrm{n_{max}})$.

4.3 | The Micro-tuning Strategy

As hinted, in our experimental study we observed that, once fixed a configuration, the power usage can be subject in various cases to small variations around its average value. Basically, this is due to the fact that the instruction paths executed by cores can change over time, and that different instructions may require different power usage (see [40]). Thus, once the selection strategy (either exploration-based or model-based) has selected the optimal configuration—to be ideally kept up to the next execution of the selection phase—if the instruction path temporarily changes in a way to demand more power, then power cap violations are likely to happen. In particular, this may happen if the power usage of the selected configuration is very close to the power cap. On the other hand, if the instruction path temporarily changes in a way to demand less power, then there could be the possibility to move, without violating the power cap, to a power state with higher frequency for improving performance. To cope with these small and temporarily power usage variations, we designed a procedure that implements a micro-tuning strategy, which is activated immediately after a configuration is set by the selection strategy. The micro-tuning strategy considers the two power states adjacent to the one chosen by the selection strategy, i.e., one with lower frequency and one with higher frequency. Upon a power cap violation, the micro-tuning strategy switches to the configuration with lower frequency. Conversely, when the power usage decreases, the procedure tries to achieve a performance boost by switching to higher core frequency. This setting is kept along time unless the power usage violates the power cap. Upon any violation, the micro-tuning strategy switches the core frequency back to the previous value.

Figure 5 shows an example timeline for the exploitation of our selection strategies (exploration-based or model-based) in combination with the micro-tuning strategy. As depicted, the selection strategy is periodically re-executed, and the micro-tuning strategy runs between two consecutive executions of the selection strategy.

5 | EXPERIMENTAL STUDY

In this section, we present the results of the experimental study we conducted to assess the proposed power capping technique. As in previous studies (e.g. [15,41]), we consider two metrics, i.e. the average power cap error and the speed-up of the application. Preliminary, we present the results of the study we performed to estimate the prediction accuracy of the two models used by the model-based strategy. Then, we present the results of the evaluation study of our power capping technique, comparing it with the following two techniques taken from the literature:

1. A technique that uses the selection strategy proposed by Reda et al.[15], which has been already used as a reference in various studies. The strategy identifies the highest number of cores for which at least one admissible configuration exists. Then selects, within this set of configurations, the one with the highest frequency P-state. We refer to this technique as baseline.

2. A technique based on a strategy that executes an hill-climbing search to find the number of cores which provides the maximum throughput at the lowest frequency P-state. Then, it increases P-state until the power usage is below the power cap. We refer to this strategy as dual-phase. It is a refinement of the technique proposed by Zhang et al.[16].

The comparison with the baseline technique allows us to quantify the performance benefits achievable by taking into account the scalability profile of the application, as we do in our technique. The comparison with the dual-phase technique, which accounts for the scalability profile, allows us to quantify the performance benefit of our technique compared to a solution based on two distinct mono-dimensional exploration phases. Our experimental study shows that such a solution can offer acceptable performance levels. However, in general, it can fail in the identification of the optimal configuration, thus our technique can offer better performance.

In our experiments, we considered three different groups of applications:

- **Group A**, which includes applications with highly scalable workload, reflecting Profile 1.
- **Group B**, which includes applications with scalability profiles spanning from Profile 2 to Profile 3.
- **Group C**, which includes applications with workloads showing extremely limited scalability, reflecting Profile 4.

The first one (Group A) is representative of applications for which the highest performance is achieved when using the highest number of available cores. The second one (Group B) is representative of applications for which the highest performance is achieved when the number of used cores is in the middle, i.e. between the minimum and maximum number of available cores. Finally, the third one (Group C) is representative of applications for which the highest performance is achieved using just one core. This also explains why we included applications with Profile 2 and Profile 3 in Group B.

Based on the analysis we carried out on the scalability profile of applications taken from various benchmarks, which we ran on the 20-core machine exploited in our study (see Section 3 for the details), we selected a total of twelve different applications, four for each group. As for Group A, we selected four applications from NAS Parallel Benchmark[27]. They are: Block tridiagonal solver, 3-D Fft Pde, Multigrid and Pentadiagonal solver. We note that applications in Group A can be considered as worst-cases when comparing our technique to the baseline. In fact, since all of them definitely scale up to the number of available cores, the the baseline can easily find the optimal configuration. At the same time, the baseline does not pay the overhead for supporting the strategies used in our power capping technique. Also, we observed that, fixed a configuration, the power usage profile of these applications is very stable along the whole application execution. Thus, these applications can be considered as worst-case test-beds also with respect to the micro-tuning strategy, since there is no room for providing fine grained optimizations based on this strategy.

As for Group B, we selected four applications from the STAMP benchmark suite[28], namely Vacation, Intruder, Genome and Ssca2, with their TM-based implementations. Their scalability profiles well match Profile 2 and Profile 3. Finally, as for Group C, we found that the implementations of the above mentioned STAMP applications where the critical sections are synchronized with locks rather than transactions—as it occurs in the STM based implementation—represent good test cases matching Profile 4. This is due to the fact that coarse-grained lock-based synchronization is generally less scalable than the fine-grained transactional approach.

With the Intel TurboBoost option (or similar ones such as AMD Turbo CORE) the real power usage and the clock frequency of a core may become highly variable. If fact, they may run out of the control of the Operating System, being regulated by the hardware on the basis of various factors, such as the current power consumption of the other cores in the same package, and the overall package temperature. Accordingly, they can become unpredictable in practice. For this reason, the P-state that enables Intel TurboBoost (which is typically P0) should be excluded to guarantee to avoid uncontrolled power cap violation. However, on those machines which suffer from this problem (as the one used in our experimental study) our technique can circumvent it, without loosing the possible performance advantage of offered by P0. This can be simply done by excluding P0 from the set of P-states considered by the decision strategy. P0 still remains reachable by the micro-tuning strategy, which can switch to P0 when P1 has been selected. If with P0 the power consumption is violated, the micro-tuning strategy immediately switches back to P1. Definitely, our technique is able to exploit Intel TurboBoost when the power cap value does not lead to disabling it.

In the next sections we initially focus on implementation details of our power capping technique. Then we present and discuss the experimental results.

5.1 | Implementation Details

All software components of our experimental study, including the benchmark applications, are written using the C language and Linux/Posix services[2]. We designed a controller module that periodically runs the selection procedure, at the end of which it sets the selected configuration. Once a configuration is set, the controller runs the procedure that implements the micro-tuning strategy until the subsequent re-execution of the selection procedure. The selection procedure is re-started after a period of time which lasts 10 times the duration of the selection procedure. We observed that this choice offers a good trade-off between the overhead introduced by the selection procedure and the optimal solution search frequency. We will discuss this aspect more in detail later in this section.

In our implementation, the power usage is measured via the RAPL interface, and the P-state is regulated through *cpufreq* Linux sub-system, which in recent versions of the Linux kernel can be accessed through the */sys* virtual file system. We observed that, with the machine used in our experiments, the time to execute the software instructions to change the frequencies of all cores through this interface is below 1 millisecond. We note that, with our technique, the overhead to execute these instructions is paid only by one core (which is in charge of changing the P-states of all the other cores), and only when it is required to actually move to a new configuration with a different P-state with respect to the previous configuration. Finally, we observed that the hardware transition delay to move from a frequency to another was in the order of tens of microseconds.

As mentioned in Section 1, our technique aims at maximizing a performance metric which is directly correlated to the actual progress of the application. The target metric we selected for the presentation of our power capping technique, namely the throughput, well fits all the application benchmarks considered in this study. In NAS applications, which run computations based on iterative loops, the throughput can be simply measured in terms of number of iteration steps executed per time unit. In STAMP applications, the throughput can be measured in terms of transactions or critical sections executed per time unit.

In NAS applications we applied our power capping technique by relying on the core-packing approach. Specifically, the controller forces all threads (by changing the threads' scheduling affinity via the function *pthread_setaffinity_np*) to run on the number of cores decided by our technique. With STAMP applications, we used an alternative scheme in which the controller changes the number of used cores by pinning threads to different cores, and by activating/deactivating the different threads to match the selected number of cores. Threads are deactivated by calling the *pause()* service and Posix signals are used to wake up the threads again.

With the machine we used in our experiments, we noticed that the minimum length of the sampling interval to get reliable measurements of the power usage is about 50 ms. As for the throughput measurements with the selected applications, we observed that a duration in the order of a few milliseconds was sufficient. Consequently, we used 50 ms as a lower bound for the duration of each measurement step of the selection procedure.

Based on these settings, the actual duration of the selection procedure varied depending on the application and the selection strategy, being it directly correlated to the number of explored configurations. With our 20-core machine, the model-based strategy has a number of exploration steps equal to 40 (see Section 4.2.3), with an average duration of about 2 seconds. Differently, the number of steps required by the baseline technique, the dual-phase technique and the exploration-based strategy change depending on the shape of the throughput curves. In all our experiments, in the majority of cases the baseline and the dual-phase techniques required between 10 and 30 measurement steps (with a duration between 0,5 and 1,5 seconds), the exploration-based strategy required between 10 and 50 measurement steps (with a duration between 0,5 and 2,5 seconds) for the first execution, and between 10 and 30 measurement steps (with a duration between 0.3 and 0,8 seconds) for the subsequent executions [3]

5.2 | Model Prediction Accuracy

We evaluated the prediction accuracy of the two analytical models used by the model-based strategy for all the twelve target applications. To instantiate the models, we measured the throughput and the power usage while varying the number of exploited cores along the execution of the applications from 1 to 20 and for two different P-state values. We selected the lowest and the highest P-State, namely *P11* and *P1*. Then, once instantiated the models using the collected measures, we used the models to estimate the throughput and the power usage for all the other $(\mathsf{p}_{\mathsf{max}} - 2) \cdot \mathsf{n}_{\mathsf{max}}$ configurations (i.e. $9 \cdot 20 = 180$ configurations), and we compared them with the measured values. The Mean Absolute Percentage Error (MAPE) is reported in Table 1 for each application.

The results show that the error is always below 5%. The maximum error of the power-usage model is 4,81%, and the maximum error of the performance model is 4.61%. Also, there is no significant difference among the errors of the models when working with applications in the three different groups. These results demonstrate both a good accuracy of the analytical models and a good robustness, since their accuracy appears to be independent of the type of the workload and scalability profile.

[2]Software is publicly available at *https://github.com/HPDCS/NAS-powercap* and *https://github.com/HPDCS/STAMP-powercap*

[3]The difference in the duration of the first execution and the subsequent executions of the selection procedure with the exploration-based strategy is due to the fact that the first execution start from a random configuration, while the subsequent executions start from the configuration selected during the last execution, which is likely closer to the optimal one, unless the workload is remarkably varied.

Group	Test case	Power-usage Model Prediction error	Performance Model Prediction error
A	Block tridiagonal solver	3.12%	4.61%
	CG	2.92%	4.08%
	3-D Fft Pde	2.91%	4.14%
	Multigrid and Pent.	3.16%	2.58%
B	Genome with TM	2.42%	2.93%
	Intruder with TM	4.82%	2.61%
	Ssca2 with TM	3.07%	2.43%
	Vacation with TM	4.81%	2.98%
C	Genome with Locks	2.26%	3.29%
	Intruder with Locks	3.41%	2.91%
	Ssca2 with Locks	2.02%	2.04%
	Vacation with Locks	3.25 %	2.21%

Table 1 Mean Absolute Percentage Error (MAPE) of the performance model and the power-usage model.

5.3 | Evaluation and Comparison with Other Techniques

With all cores running at the maximum power, the power usage of the machine we used in our experiments is about 90 watts. Accordingly, we used two different power cap values, namely 50 watts and 70 watts. We consider two evaluation metrics, the power cap violation percentage and the throughput speed-up. The power cap violation percentage is calculated as the average of the percentage difference between the measured power usage and the power cap value for each time interval of 1 second. For those time intervals for which the power cap is not violated the percentage is considered equal to zero. The speed-up of the throughput is measured with respect to the throughput of the baseline. We measured the above-mentioned metrics for the baseline, the dual-phase technique, and our power capping technique with either the exploration-based strategy and the model-based strategy.

The results in the top chart of Figure 6 show the power cap violation and the speed-up with power cap set to 50 watts. Let us consider the four test cases of Group A. We remark that they represent worst-case tests for our technique because their workload profiles scale very well. Also, the instruction path of these applications is extremely uniform over time, and gives rise to very small variations of the power usage. Consequently, there is little room for the micro-tuning strategy to provide additional performance boosts. Overall, test cases in Group A are interesting mostly for the observation of the overhead induced by our solution in scenarios where scarce possibilities to provide optimizations (compared to the baseline) exist.

Figure 6 shows that our technique provides slow-down (i.e. the throughput is lower than the baseline) with only one of the four test cases of Group A, specifically "Black Tridiagonal Solver". However, such slow-down is no more than 5% with the exploration-based strategy and 3% with the model-based strategy. On the other hand, the dual-phase technique shows a slow-down of about 30%. With the other three test cases of Group A, our technique offers speed-up of up to 10%. These results show that the overhead introduced by the technique is affordable. Also, our technique consistently performs better than the dual-phase technique. The power cap violation percentage with our technique with the exploration-based strategy is higher than the baseline only with "3-D Ftp Pde". However, its value is always below 1%. With the model-based strategy the violation percentage is always higher than the baseline. However, also in this case it is limited to 1%, except for "Multigrid" where the value is of about 2%.

As for Group B, the dual-phase technique and our technique (with both the strategies) offer a noticeable increment of the speed-up for "SSca2 with TM" and "Intruder with TM". This is motivated by the scalability profile of the two applications, which matches Profile 3, while the scalability profile of "Vacation with TM" and "Genome with TM" matches Profile 2. Our technique with the heuristic-based strategy achieves higher speed-up increments (up to 85% increased speed-up over baseline for Intruder) and, independently of the employed selection strategy, it significantly outperforms the dual-phase technique for 2 out of 4 test cases—with up to 35% better speed-up for Scaa2—while providing the same performance for the other 2 test cases. As for the power cap, with Group B the percentage violation is still relatively low, being under 1% with all the tested techniques. On the other hand, the results confirm that our technique used with the model-based strategy leads to a violation percentage that is generally higher, although it is anyhow limited to 2% over all test cases. This is essentially due to the prediction error of the power-usage model[4]. In fact, the violation percentages are in line with the model prediction errors we measured (see Table 1).

[4]Even if the prediction error is very low, it could lead to select some configuration that is believed to be within the power cap, but it is actually not.

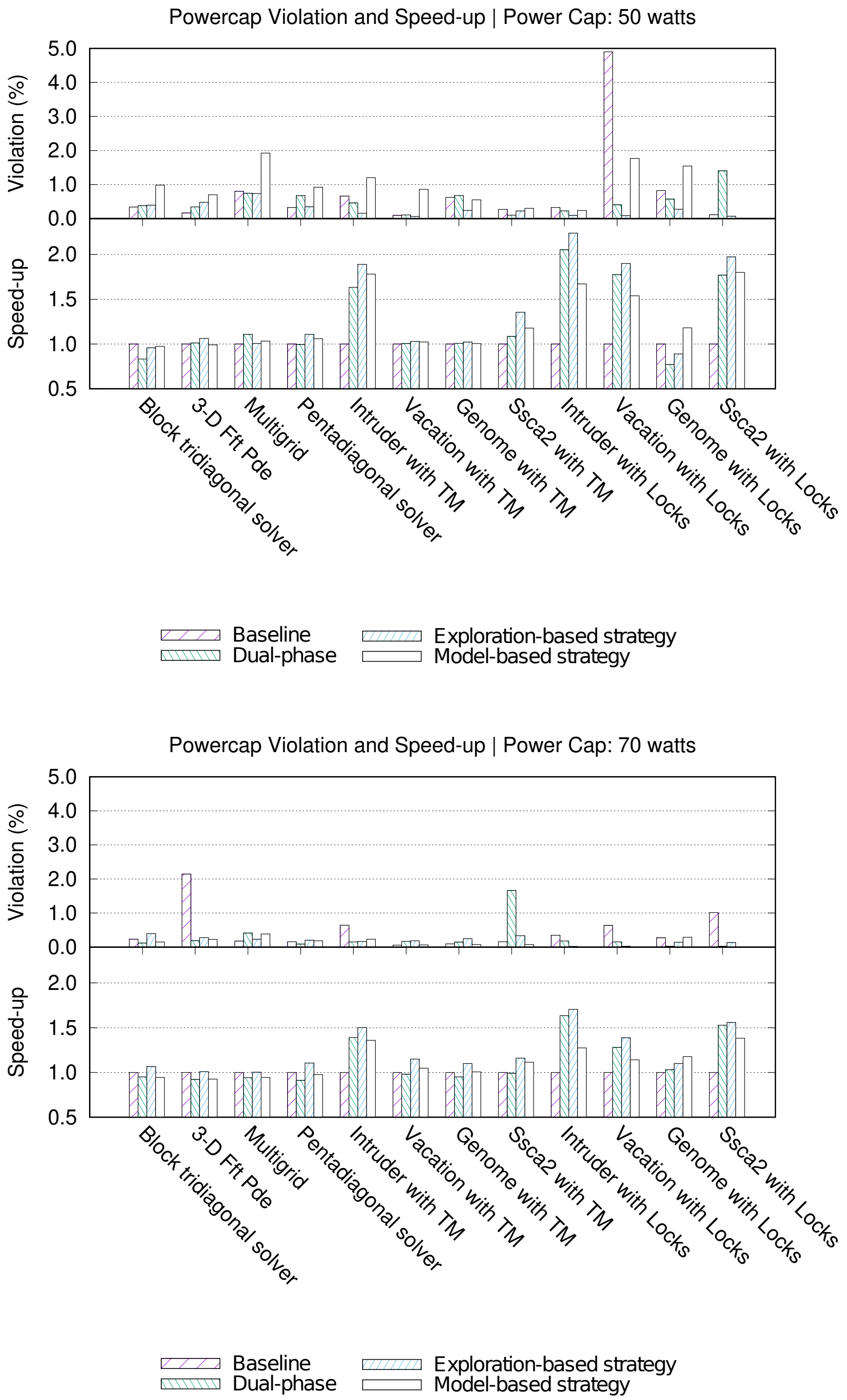

Figure 6 Power cap violation and speed-up results

Finally, as for Group C, the dual-phase technique and our technique with the exploration-based strategy improve the speed-up with all the test cases, except "Genome with Locks". For this test case, the model-based strategy works better than the exploration-based strategy. We observed that this is due to the scalability profile of Genome, which is quite irregular and shows more than one local maximum. This represents a disadvantage for the dual-phase technique and the exploration-based strategy of our technique, which may fall in a local maximum which could not be the global one. Differently, the model-based strategy does not suffer from this limitation, thanks to the models which allow to get predictions of the power

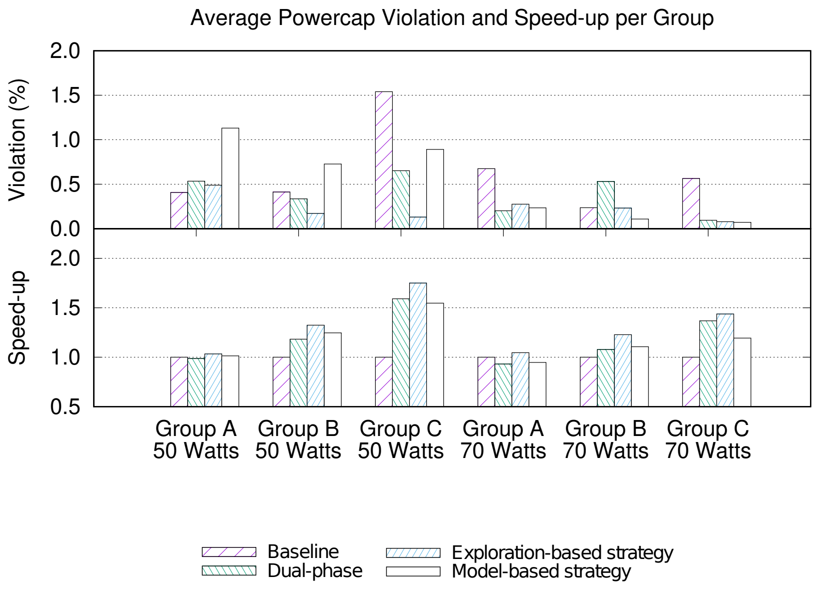

Figure 7 Average Power cap violation and speed-up per group

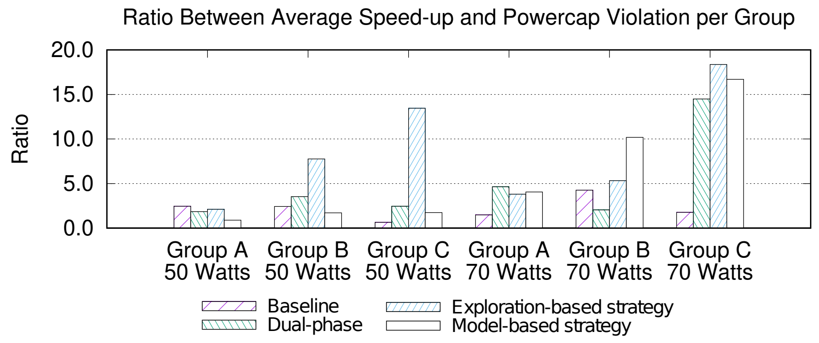

Figure 8 Ratio Between speed-up and average Power cap violation per group

usage and the speed-up values for the whole configuration space. In any case, with Group C, our technique outperforms the dual-phase technique, providing speed-up improvements of up to 9% for "Scaa2 with locks" and up to 100% for "Genome with locks".

The results in the bottom chart of Figure 6 show the power cap violation and the speed-up with power cap set to 70 watts. The test cases of Group A confirm that the overhead introduced by our technique is very low. It is slightly higher compared to the results with power cap equal to 50 watts, in particular those achieved with the model-based strategy. This is caused by the larger set of admissible configurations when the power cap is equal to 70 watts with respect to 50 watts, requiring more time for the exploration and for building the models. However, the speed-up provided by our technique is lower than the one of the baseline only when used with the model-based strategy; the difference is anyhow limited to 8% The speed-up values with the test cases of Group B and Group C essentially confirm the results we observed with power cap equal to 50 watts. Also, in these cases the exploration based strategy achieves the highest speed-up, with the only exclusion of "Genome with Locks", for which the model-based strategy is the best one. As for the power cap violation percentage, with cap of 70 watts it is (on average) lower compared to cap of 50 watts. This is because of the less strict constraint on the set of admissible configurations imposed with cap of 70 watts. In fact, when this set is larger, the probability that the power usage of the optimal configuration is far from the power cap is generally higher, thus reducing the overall violation probability.

5.4 | Overall Assessment and Discussion

To make an overall assessment of the results of our experimental study, in Figure 7 we report the calculated average power cap violation percentage and the average speed-up for each group of applications. Basically, the results clearly demonstrate the advantages, in terms of speed-up, achievable with techniques that take into account the scalability of the applications. In particular, the advantages become higher when moving from scalable workloads to less scalable ones, for which smarter approaches that avoid over-provisioning of cores in combination with the fine tuning of the core power states have more chances to pay off. Further, the results show that our technique with the exploration-based strategy consistently outperforms the dual-phase technique. Basically, this is due to the ability of the exploration-based strategy to guarantee, under the same assumptions of the dual-phase technique, to find the optimal configuration for a superset of cases. As for the model-based strategy, the results show that it outperforms the baseline with workloads with limited scalability. Further, it outperforms the dual-phase technique and the exploration-based strategy when the workload profile is irregular, where the unimodality assumption generally may not hold. In conclusion, in scenarios with irregular scalability profiles, although the prediction error of the model-based strategy may be non-negligible, the model-based strategy can offer better results.

We conclude our assessment study by analysing some additional data, in order to exclude that the performance advantages offered by the strategies we proposed could be due in some way to the extra power they use in those cases where the power cap violation is higher than the baseline. To this aim, in Figure 8 we report the ratio between the speed-up and the average power cap violation for each group of applications. The histograms show that with the exploration-based strategy and with the model-based strategy the ratio is higher than the baseline in the most of cases, except for Group A 50 Watts, and for Group B 50 Watts only with the model-based strategy. We remark that Group A represents worst-case scenarios for our technique in terms of performance maximization. In all the other cases, the violation with the exploration-based strategy is lower than the baseline, and the exploration-based strategy achieves higher speed-up. Also, we note that the model-based strategy offers higher performance than the baseline not only with Group B 50 Watts, but also with Group B 70 Watts, with Group C 50 Watts and with Group C 70 Watts. In these cases, the violation with the model-based strategy is lower than the baseline. Accordingly, these data indicate that the performance improvement with both the strategies is not generally correlated to the additional power usage which in some cases can be exploited by the strategies with respect to the baseline.

In conclusion of our study, we want briefly discuss about the applicability of our approach and its possible limits. To optimise the performance of a given application, it is necessary to have the possibility to measure an application-specific performance metric, such as the throughput or the response time, or some other metric that measures the execution progress of the application. Considering our target, namely performance-sensitive multi-threaded applications, they often expose performance metrics to the user to monitor their actual level of performance. As an example, in applications that process transactions, web requests or batch processes, measuring the throughput is simple and is already allowed by specific frameworks. On the other hand, there may be applications in which the performance metrics could be not directly accessible. This would require to insert within the application code some code block to measure the target performance metric. This would be more easy to do when the application source code is available. Conversely, when the source code is not accessible, it would be necessary to try to use some different approach, e.g by wrapping specific components of the application to measure the desired performance metric.

Another issue to consider is related to applications that may show some irregular behaviour at run-time which may cause continuous variation of the values of the target performance metric. This could be the case of applications where, e.g. the instruction execution path quickly and irregularly changes over time. Obviously, frequent and fast changes in the workload of the applications can represent a weak point for adaptive techniques. Indeed, in these cases the optimal solution may continuously change over times. It might even change so fast that the search of the optimal solution may never converge. We empirically observed that when the workload of the application is subject to continuous changes, such that the performance and the power usage are generally not stable for a time period of about 10 times the duration of the selection procedure (see Section 5.1), the efficiency of our technique may start decreasing. On the other hand, this kind of irregular behaviours generally disfavours not only our technique, but rather any adaptive approach.

6 | CONCLUSIONS

In this article, we investigated the design of power capping techniques for multi-thread applications. We explored both a heuristic approach and a model-based approach for designing on-line strategies that tune at run-time the number of cores used by the application and their power state. Our study suggests that strategies designed to account for the scalability profile of the application can offer noticeable advantages in terms of performance. From a theoretical perspective, the exploration-based strategy we designed can find the optimal configuration in linear time, under the assumption of unimodality of the workload scalability profile. The experimental results showed that such kind of strategy works well with a variety of test cases, thus suggesting that a heuristic exploration-based approach can represent a viable solution. On the other hand, when the scalability profile markedly deviates from the unimodality assumption, it would be convenient to use model-based approaches, like the one we presented. Ultimately, the strategies we presented should be considered as alternatives to be chosen on the basis of the workload.

References

1. Avgerinou Maria, Bertoldi Paolo, Castellazzi Luca. Trends in Data Centre Energy Consumption under the European Code of Conduct for Data Centre Energy Efficiency. *Energies.* 2017;10:1470.
2. Buyya Rajkumar, Vecchiola Christian, Selvi S. Thamarai. *Mastering Cloud Computing: Foundations and Applications Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1st ed.2013.
3. Oracle . Plug into the Cloud with Oracle Database 12C (wite paper) http://www.oracle.com/technetwork/database/plug-into-cloud-wp-12c-1896100.pdf.
4. Vitali Roberto, Pellegrini Alessandro, Quaglia Francesco. Load Sharing for Optimistic Parallel Simulations on Multi-core Machines. *ACM Performance Evaluation Review.* 2012;43(3):2–11.
5. Pallipadi Venkatesh, Starikovskiy Alexey. The ondemand governor: past, present and future. In: *Proceedings of Linux Symposium, vol. 2, pp. 223-238*; 2006.
6. Su Bo, Gu Junli, Shen Li, Huang Wei, Greathouse Joseph L., Wang Zhiying. PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture.* 2014;:445–457.
7. Liu Yanpei, Cox Guilherme, Deng Qingyuan, Draper Stark C., Bianchini Ricardo. FastCap: An efficient and fair algorithm for power capping in many-core systems. *ISPASS 2016 - International Symposium on Performance Analysis of Systems and Software.* 2016;(3):57–68.
8. Conoci Stefano, Di Sanzo Pierangelo, Ciciani Bruno, Quaglia Francesco. Adaptive Performance Optimization under Power Constraint in Multi-Thread Applications with Diverse Scalability. In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*ICPE ’18:16–27Association for Computing Machinery; 2018; New York, NY, USA.
9. David Howard, Gorbatov Eugene, Hanebutte Ulf R., Khanna Rahul, Le Christian. RAPL: Memory power estimation and capping. *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on.* 2010;:189–194.
10. Muthukaruppan Thannirmalai Somu, Pricopi Mihai, Venkataramani Vanchinathan, Mitra Tulika, Vishin Sanjay. Hierarchical power management for asymmetric multi-core in dark silicon era. *Proceedings of the 50th Annual Design Automation Conference on - DAC ’13.* 2013;:1.
11. Ryoo Shane, Rodrigues Christopher I., Baghsorkhi Sara S., Stone Sam S., Kirk David B., Hwu Wen-mei W.. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*PPoPP ’08:73–82ACM; 2008; New York, NY, USA.
12. Hellerstein Joseph, Morrison Vance, Eilebrecht Eric. Applying control theory in the real world: Experience with building a controller for the .NET thread pool. *SIGMETRICS Performance Evaluation Review.* 2009;37:38-42.
13. Di Sanzo Pierangelo. Analysis, Classification and Comparison of Scheduling Techniques for Software Transactional Memories. *IEEE Transactions on Parallel and Distributed Systems.* 2017;28(12):3356-3373.
14. Ianni Mauro, Marotta Romolo, Cingolani Davide, Pellegrini Alessandro, Quaglia Francesco. The Ultimate Share-Everything PDES System. In: *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*SIGSIM-PADS ’18:73–84ACM; 2018; New York, NY, USA.
15. Reda Sherief, Cochran Ryan, Coskun Ayse. Adaptive Power Capping for Servers with Multithreaded Workloads. *IEEE Micro.* 2012;32(5):64–75.
16. Zhang Huazhe, Hoffmann Henry. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*ASPLOS ’16:545–559ACM; 2016; New York, NY, USA.

17. Porterfield Allan K., Olivier Stephen L., Bhalachandra Sridutt, Prins Jan F.. Power measurement and concurrency throttling for energy reduction in OpenMP programs. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*. 2013;:884–891.

18. Intel . *Intel 64 and IA-32 Architectures Software Developer Manual, Volume 3C: System Programming Guide, Part 3*. 2011.

19. Gholkar Neha, Mueller Frank, Rountree Barry. Power Tuning HPC Jobs on Power-Constrained Systems. *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation - PACT '16*. 2016;:179–191.

20. Bari Md Abdullah Shahneous, Chaimov Nicholas, Malik Abid M., et al. ARCS: Adaptive runtime configuration selection for power-constrained OpenMP applications. *Proceedings - IEEE Int. Conference on Cluster Computing, ICCC*. 2016;:461–470.

21. Dennis JE, Woods Daniel J. Optimization on microcomputers: The Nelder-Mead simplex algorithm. *New computing environments: microcomputers in large-scale computing*. 1987;11:6–122.

22. Deng Qingyuan, Ramos Luiz, Bianchini Ricardo, Meisner David, Wenisch Thomas. Active low-power modes for main memory with memScale. *IEEE Micro*. 2012;32(3):60–69.

23. David Howard, Fallin Chris, Gorbatov Eugene, Hanebutte Ulf R, Mutlu Onur. Memory Power Management via Dynamic Voltage/Frequency Scaling. *Proceedings of the 8th ACM International Conference on Autonomic Computing*. 2011;:31–40.

24. Kanduri Anil, Haghbayan Mohammad-Hashem, Rahmani Amir M., et al. Approximation knob: power capping meets energy efficiency. *Proceedings of the 35th International Conference on Computer-Aided Design - ICCAD '16*. 2016;:1–8.

25. Li Bo, León Edgar A., Cameron Kirk W.. COS: A Parallel Performance Model for Dynamic Variations in Processor Speed, Memory Speed, and Thread Concurrency. In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2017, Washington, DC, USA, June 26-30, 2017*:155–166; 2017.

26. Corporation Compaq Computer, B Revision. *Advanced Configuration and Power Interface Specification*. 2000.

27. Bailey David H.. NAS Parallel Benchmarks:1254–1259. Boston, MA: Springer US 2011.

28. Cao Minh Chi, Chung JaeWoong, Kozyrakis Christos, Olukotun Kunle. STAMP: Stanford Transactional Applications for Multi-Processing. In: *Proc. 4th IEEE Int. Symposium on Workload Characterization*:35–46IEEE; 2008.

29. Felber Pascal, Fetzer Christof, Riegel Torvald. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*; 2008.

30. Daloze Benoit, Tal Arie, Marr Stefan, Mössenböck Hanspeter, Petrank Erez. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. *Proc. ACM Program. Lang.*. 2018;2(OOPSLA):108:1–108:30.

31. Pllana S., Xhafa F.. *Programming Multicore and Many-core Computing Systems*. Wiley Series on Parallel and DWiley; 2017.

32. Xi Bowei, Liu Zhen, Raghavachari Mukund, Xia Cathy H., Zhang Li. A Smart Hill-climbing Algorithm for Application Server Configuration. In: *Proceedings of the 13th International Conference on World Wide Web*WWW '04:287–296ACM; 2004; New York, NY, USA.

33. Schaefer Christoph A.. Reducing Search Space of Auto-tuners Using Parallel Patterns. In: *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*IWMSE '09:17–24IEEE Computer Society; 2009; Washington, DC, USA.

34. Kirkpatrick S., Gelatt C. D., Vecchi M. P.. Optimization by Simulated Annealing. *Science*. 1983;220(4598):671–680.

35. Glover Fred. Future Paths for Integer Programming and Links to Artificial Intelligence. *Comput. Oper. Res.*. 1986;13(5):533–549.

36. Stützle Thomas, Ruiz Rubén. Iterated Local Search:579–605. Cham: Springer International Publishing 2018.

37. Nath Rajib, Tullsen Dean. The CRISP Performance Model for Dynamic Voltage and Frequency Scaling in a GPGPU. In: *Proceedings of the 48th International Symposium on Microarchitecture*MICRO-48:281–293ACM; 2015; New York, NY, USA.

38. Rauber Thomas, Rünger Gudula, Schwind Michael, Xu Haibin, Melzner Simon. Energy Measurement, Modeling, and Prediction for Processors with Frequency Scaling. *J. Supercomput.*. 2014;70(3):1451–1476.

39. Kaxiras Stefanos, Martonosi Margaret. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers; 1st ed.2008.

40. Hirki Mikael, Ou Zhonghong, Khan Kashif Nizam, Nurminen Jukka K., Niemi Tapio. Empirical Study of the Power Consumption of the x86-64 Instruction Decoder. In: *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16)*USENIX Association; 2016; Santa Clara, CA.

41. Lefurgy Charles, Wang Xiaorui, Ware Malcolm. Power Capping: A Prelude to Power Shifting. *Cluster Computing*. 2008;11(2):183–195.

How to cite this article: Stefano Conoci, Pierangelo Di Sanzo, Alessandro Pellegrini, Bruno Ciciani, and Francesco Quaglia (2020), On Power Capping and Performance Optimization of Multi-threaded Applications, *Concurrency and Computation: Practice and Experience*, *2020;00:1–6*.

APPENDIX

A PROOF OF THE OPTIMALITY OF THE CONFIGURATION SELECTED BY THE EXPLORATION-BASED STRATEGY

In this section, we prove that the exploration-based strategy finds the optimal configuration—although it does not explore the whole configuration space—under the assumption of unimodality of the speed-up curve and the assumption that the power usage always increases when decreasing P-state or when increasing the number of used cores (which actually matches what we have experimentally observed).

We can formalize the following derived assumption, which are functional to the construction of the optimality proof.

Assumption 1. Fixed a P-state, the speed-up curve while varying the number of used cores n in $[1, \mathsf{n}_{\mathsf{max}}]$ shows one of the following trends:

1. initially increases, reaches its maximum value, then decreases, otherwise
2. monotonically increases, otherwise
3. monotonically decreases.

Assumption 2. If $\mathsf{thr}(\mathsf{p}', \mathsf{n}) > \mathsf{thr}(\mathsf{p}', \mathsf{n}+1)$ then for each p we have $\mathsf{thr}(\mathsf{p}, \mathsf{n}) > \mathsf{thr}(\mathsf{p}, \mathsf{n}+1)$. Also, if $\mathsf{thr}(\mathsf{p}', \mathsf{n}) > \mathsf{thr}(\mathsf{p}', \mathsf{n}-1)$ then for each p we have $\mathsf{thr}(\mathsf{p}, \mathsf{n}) > \mathsf{thr}(\mathsf{p}, \mathsf{n}-1)$. In other words, if for some P-state and n cores the speed-up decreases (increases) when adding (removing) one core, then this holds true for whichever P-state. Overall, the ordering relations on the speed-up values when changing the number of cores are not effected by P-state.

Assumption 3. If $\mathsf{p} < \mathsf{p}'$ then $\mathsf{thr}(\mathsf{p}, \mathsf{n}) > \mathsf{thr}(\mathsf{p}', \mathsf{n})$ for whichever n. In other words, when decreasing the value of P-state the speed-up always increases for whichever number of cores.

Assumption 4. If $\mathsf{p} < \mathsf{p}'$ then $\mathsf{pwr}(\mathsf{p}, \mathsf{n}) > \mathsf{pwr}(\mathsf{p}', \mathsf{n})$, and if $\mathsf{n} > \mathsf{n}'$ then $\mathsf{pwr}(\mathsf{p}, \mathsf{n}) > \mathsf{pwr}(\mathsf{p}, \mathsf{n}')$. In other words, the power usage increases when decreasing P-state or when increasing the number of cores.

Theorem 1. The exploration-based strategy finds the configuration with the maximum speed-up and that does not violate the power cap.

Proof. We partition the search space into three disjoint sub-spaces, based on the value of P-state of the initial configuration, i.e. p^{s}. Specifically:

- S_1 is the sub-space of configurations such that $\mathsf{p} = \mathsf{p}^{\mathsf{s}}$;
- S_2 is the sub-space of configurations such that $\mathsf{p} < \mathsf{p}^{\mathsf{s}}$;
- S_3 is the sub-space of configurations such that $\mathsf{p} > \mathsf{p}^{\mathsf{s}}$.

We show that Phase 1, Phase 2 and Phase 3 find the optimal configuration for sub-spaces S_1, S_2 and S_3, respectively. This is sufficient to prove that the overall optimal configuration is found, since Final phase simply selects the optimal one among them.

Outcome of Phase 1.

Phase 1 explores configurations within S1. Specifically, it keeps fixed p^{s} and explores while varying the number of cores n. Phase 1 uses the hill-climbing search. By Assumption 1, the function $\mathsf{thr}(\mathsf{p}^{\mathsf{s}}, \mathsf{n})$ has only one local maximum, which corresponds to the global maximum. Accordingly, the hill-climbing search can trivially find the maximum, which is the optimal configuration in S_1. The only exception is when the configuration with the global maximum violates the power cap. In this case, the exploration terminates as soon as the configuration with the highest number of cores, which is within the power cap, is found. Also in this case, it is the optimal configuration in S_1

Outcome of Phase 2.

We recall that Phase 2 starts exploring from the configuration returned by Phase 1, denoted as (p^s, n^1), which is the optimal among the ones with P-state equal to p^s, unless none of them is within the power cap. In the latter case, Phase 1 returns $(p^s, 1)$, i.e. the one with the lowest power consumption. Also, we recall that Phase 2 explores moving towards lower P-states and a lower number of cores. By Assumption 2, the number of cores that provides the maximum speed-up does not change when decreasing P-state. Accordingly, if (p^s, n^1) is the optimal configuration fixed p^s, then the optimal configuration for the sub-space S_2 must have a number of cores less than or equal to n^1. Specifically, if $\mathsf{pwr}(p^s - 1, n^1) < C$ then the optimal configuration with P-state equal to $p^s - 1$ still has n^1 cores. Otherwise, if the power cap is violated, the number of cores has to be reduced to try to stay within the power cap (by Assumption 4). This leads to reduce the speed-up, since the above situation can arise only if we are in the ascending part of the speed-up curve. Accordingly, in this case the optimal configuration is the first one that is within the power cap while reducing the number of cores. Phase 2 follows exactly this behavior, i.e. it first moves to P-state equal to $p^s - 1$, and if $\mathsf{pwr}(p^s - 1, n^1) > C$ then it reduces the number of cores until it finds a configuration that does not violate the power cap. Thus, Phase 2 finds the optimal configuration for P-state equal to $p^s - 1$, unless none of them is within the power cap. We remark that Phase 2 continues this search for each P-state such that $p \in [1, p^s - 1]$. Thus, it finds the optimal configuration for each P-state in the sub-space S_2. Ultimately, it selects the optimal one of them, thus finding the optimal configuration in the sub-space S_2.

Outcome of Phase 3.

We remark that Phase 3 starts exploring from the configuration returned by Phase 1, and explores moving towards higher P-state values and a higher number of cores. Also, we remark that Phase 3 is not executed if the configuration returned by Phase 1 is such that n^1 is the number of cores that provides the highest speed-up and is within the power cap. Indeed, in this case, the speed-up for any configuration with a number of cores higher than n^1 and any higher P-state is lower by Assumption 2. Hence, Phase 3 is executed only if the number of cores that provides the highest speed-up is higher than n^1, but it violates the power cap with P-state equal to p^1. This means that n^1 is along the ascending part of the speed-up curve because of Assumption 1. Also, this holds true for any P-state higher than p^1 because of Assumption 2. Accordingly, the speed-up with any configuration in the sub-space S_3 with a number of cores less than n^1 is lower. Consequently, the optimal configuration for P-state equal to $p^s + 1$ must have a number of cores higher than n^1. Phase 3 first moves to P-state equal to $p^s + 1$, then it starts increasing the number of cores (which by Assumption 4 leads to increase the power consumption) and stops when the power cap is violated or the speed-up decreases. Accordingly, it finds the optimal configuration for P-state equal to $p^s + 1$. After, Phase 3 explores for each P-state such that $p \in [p^s + 1, p_{max}]$. Thus, it finds the optimal configuration for each P-state in the sub-space S_3. Ultimately, it selects the optimal one of them, thus finding the optimal configuration in the sub-space S_3. $\square$