Adaptive Performance Optimization under Power Constraint in Multi-thread Applications with Diverse Scalability

Stefano Conoci*
DIAG – Sapienza, University of Rome
Rome, Italy
conoci@diag.uniroma1.it
conoci@lockless.it

Pierangelo Di Sanzo*
DIAG – Sapienza, University of Rome
Rome, Italy
disanzo@diag.uniroma1.it
disanzo@lockless.it

Bruno Ciciani*
DIAG – Sapienza, University of Rome
Rome, Italy
ciciani@diag.uniroma1.it
ciciani@lockless.it

Francesco Quaglia*
DICII – University of Rome Tor Vergata
Rome, Italy
francesco.quaglia@uniroma2.it
quaglia@lockless.it

ABSTRACT

Energy consumption has become a core concern in computing systems. In this context, power capping is an approach that aims at ensuring that the power consumption of a system does not overcome a predefined threshold. Although various power capping techniques exist in the literature, they do not fit well the nature of multi-threaded workloads with shared data accesses and non-minimal thread-level concurrency. For these workloads, scalability may be limited by thread contention on hardware resources and/or data, to the point that performance may even decrease while increasing the thread-level parallelism, indicating scarce ability to exploit the actual computing power available in highly parallel hardware. In this paper, we consider the problem of maximizing the performance of multi-thread applications under a power cap by dynamically tuning the thread-level parallelism and the power state of CPU-cores in combination. Based on experimental observations, we design a technique that adaptively identifies, in linear time within a bi-dimensional space, the optimal parallelism and power state setting. We evaluated the proposed technique with different benchmark applications, and using different methods for synchronizing threads when accessing shared data, and we compared it with other state-of-the-art power capping techniques.

CCS CONCEPTS

• **Hardware** → **Chip-level power issues**; *Enterprise level and data centers power issues*; • **Computer systems organization** → *Multicore architectures*; • **Software and its engineering** → *Software performance*;

KEYWORDS

Power cap; Multi-threaded workload; Performance optimization; Power efficiency

ACM Reference Format:
Stefano Conoci, Pierangelo Di Sanzo[1], Bruno Ciciani[1], and Francesco Quaglia[1]. 2018. Adaptive Performance Optimization under Power Constraint in Multi-thread Applications with Diverse Scalability. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184419

1 INTRODUCTION

Multi-core architectures are nowadays dominating the computer system market. Thanks to hardware parallelism, they offer powerful environments allowing to effectively speed-up the execution of multi-threaded workloads. These kinds of workloads span a wide range of application domains, including Web applications, transactional applications and HPC ones [15, 25]. However, one disadvantage of multi-core architectures is that powering many cores requires more energy, and power demand of computing systems raised up even more as a core concern to cope with.

Over the last years, computer system manufacturers introduced some hardware mechanisms to control power consumption and, consequently, to enable improvements in the energy efficiency. Examples include Dynamic Voltage and Frequency Scaling (DVFS), which allows lowering the voltage and the frequency (hence the power consumption) of a processor/core in a controlled manner, and Clock Gating, which disables some processor/core circuitry during idle periods. Contextually, today's Operating Systems offer power management tools—like Linux CPUFreq Governor [16]—which expose interfaces to dynamically change the power state of CPU-cores via DVFS, thus allowing to tune the performance of cores and their power consumption according to the end-users' needs.

The approach of limiting the power consumption of a system is generally known as *power capping*. In this context, an interesting challenge is the one of regulating the usage of resources, thus including power, of applications based on multi-threading technology and share data accesses. More specifically, we consider the objective of

*Also with Lockless s.r.l.

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184419

maximizing the application performance under a power constraint—the *power cap*—in scenarios where the applications themselves may exhibit different degrees of scalability. This is due to different synchronization schemes that can be used for regulating the accesses to shared data (e.g. lock-based vs. speculative ones), as well as to different incidence of conflicting accesses along the application lifetime. Overall, in these applications, part of the computing power needs to be devoted to manage synchronization (including hardware level one) which is reflected on both performance and energy consumption in non-trivial manner, as already demonstrated by a few results, e.g. [20].

In the literature various power capping techniques have been proposed (e.g. [4, 10, 14, 18, 19, 21, 24, 24]). However, most of them are application-agnostic, i.e. they enforce the power cap at the level of a server machine, without accounting for workload features (e.g. scalability) of running applications. As for the specific case of multi-threaded workloads, the problem of controlling resource usage, in terms of number of used cores and core frequency, has been addressed in some literature contributions, such as [1, 19, 26]. However, the proposed approaches suffer from some limitations. In more detail, some of them still do not account for the diverse scalability profiles of multi-threaded workloads. Other approaches rely on strategies that do not always find the best-performing configuration in terms of thread-level parallelism and frequency/voltage of used cores.

Overall, to select the right combination of thread parallelism and core power state which ensures the best performance under a power cap, it looks mandatory to take into account the (possible) limited scalability of the application, as it manifests at run-time due to synchronization dynamics. Further, it is necessary to be able to react to variations of the workload since the scalability profile of an application may change depending on the workload profile.

To cope with this problem, we propose an adaptive technique that uses a novel on-line exploration-based tuning strategy. We devised our technique exploiting empirical observations of the effects on both performance and power consumption associated with the combined variation of thread-level parallelism and CPU-core power state. Specifically, by the results of an experimental study, we show that some scalability features of multi-threaded workloads, even in the presence of non-negligible incidence of synchronization, remain invariant with respect to the variation of the power state of CPU-cores. Based on this, we defined an optimized tuning strategy where the exploration moves along specific directions that depend on the power cap value and on the intrinsic scalability of the application. Remarkably, we prove that the proposed technique finds the optimal configuration of concurrent threads and CPU frequency/voltage—the configuration that provides the highest performance among the configurations with power consumption lower than the power cap—in linear time. Also, we present a refinement of our technique that exploits fluctuations between configurations—in terms of thread-level parallelism and CPU-core power state—to further improve the application performance and reduce the possibility and the incidence of power cap violations.

We demonstrate the advantages of our proposal via an experimental study based on various application contexts, including various benchmarks that use different thread synchronization methods. This allows us to robustly assess our technique via disparate test cases where contention among threads affects the application scalability in significantly different ways.

The remainder of this article is structured as follows. In Section 2 we discuss related work. Section 3 defines our target problem and presents the results of the preliminary analysis. Section 4 illustrates the proposed optimization technique, proves that the selected configuration is optimal, analyzes the time complexity of the exploration procedure and finally presents the strategy based on fluctuations. Section 5 describes the most relevant implementation details of a software architecture embedding our optimizer, and presents the experimental results.

2 RELATED WORK

As discussed, various power capping techniques have been devised, which are aimed at limiting the overall power consumption of a single (server) machine. Most of them are based on application-agnostic approaches, thus not representing fully exhaustive solutions. Less work has been carried out targeting the reduction (or the optimization) of power consumption depending on the workload features of running applications. In the following, we focus on this kind of techniques and we discuss the differences compared to our proposal.

A work specifically focused on optimizing the power efficiency of applications with multi-threaded workloads is presented in [19]. The proposed technique, called Pack and Cap, aims at selecting the best configuration, in terms of number of CPU-cores to be assigned to an application and the related CPU-core frequency, which ensures a given power cap. Based on experimental measurements of the performance and power consumption obtained running benchmarks from the Parsec suite, the authors conclude that the configuration that provides the highest performance at a given level of power consumption always assigns to the application the highest possible number of CPU-cores. However, as shown in our experimental analysis through a direct comparison with our proposed technique, this selection strategy is not optimal for multi-thread applications with sub-linear scalability.

The work in [26] considers the problem of maximizing performance under a power cap while also taking into account the effects of thread contention on scalability. This solution defines an ordered set of power knobs that are progressively tuned by performing a binary search on the respective domain, selecting the setting that provides the highest performance for the considered power knob while operating within the power cap. Specifically, it first selects the optimal number of CPU-cores that should be assigned to an application. Subsequently, once fixed this number, it progressively increases the frequency/voltage of the CPU-cores until the power consumption is below the power cap. However, this approach may not find the configuration ensuring the best performance. Indeed, there may be some other configuration with, e.g., a lower number of threads and a higher frequency/voltage, providing higher performance still within the power cap. In our experimental analysis (see Section 5), we present data that compare this approach with our technique.

The paper by Portfield et al. [17] presents a technique that reduces the energy consumption of OpenMP programs by throttling concurrency 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 CPU-cores. This is a lightweight approach, but achieves low power reduction compared to thread pausing or modifying the frequency/voltage of CPU-cores. Nevertheless, the proposed technique does not tune the power state of the CPU-cores running non-throttled threads and does not consider power constraints.

Different literature contributions rely on Intel RAPL [4, 9] as a building block to enforce power capping at hardware level for different subsystems (e.g. CPU package or memory). They estimate the power consumption by observing different low-level hardware events, and then select the optimal CPU frequency and voltage such that the average power consumption of a specific subsystem is lower than the power cap. In our technique, we do not exploit the power capping capability offered by RAPL, although we exploit RAPL exclusively as a power measuring tool. This choice is motivated by the fact that RAPL cannot enforce power capping for whichever subset of CPU-cores. Thus, it is not adequate for application-oriented power capping, i.e. for tuning the power consumption of a specific subset of CPU-cores used by a given application, according to the fact that such number of cores well matches the scalability level of the application. Also, RAPL is a proprietary technology only supported by recent Intel x86 processors. Conversely, our technique directly controls the power state of CPU-cores via the abstraction of *P-state*, which is a standard supported by various processors from different manufacturers, also with different instruction sets.

Exploiting RAPL, Gholkar et al. [7] propose a 2-level hierarchical technique 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 between different jobs. Then, for each job, it determines the set of nodes for assigning the job, and sets the node power level via RAPL. Differently from our proposal, this technique operates at cluster-level and does not deal with thread-level parallelism. The work presented in [1] proposes an exploration-based technique that improves the performance under a power cap for OpenMP applications. For each parallel region, it selects the appropriate number of threads, scheduling policy and chunk size using the Nelder-Mead search algorithm [6]. The search is performed at a fixed RAPL 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 a-priori determined to reduce the computation time. Obviously, this approach does not guarantee to find the optimal solution.

Other works in literature investigate the problem of improving the application performance under power constraints considering different power management variables. FastCap [13] 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 considers the interaction between CPU-cores and memory banks communicating over a shared bus. Unfortunately, although memory DVFS has not been proposed recently [3, 5], it is not yet available in commercial systems.

Kanduri et al. propose to use approximation in computation as another knob that can be used in power capping, combined with DVFS and concurrency throttling, to offer a trade-off between performance and accuracy of the results [11]. Obviously, this approach

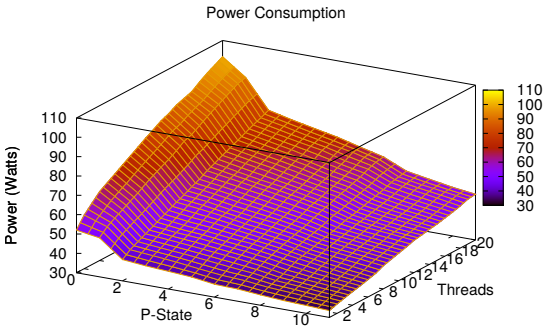

Figure 1: Throughput vs. number of concurrent threads and *P-state*

is applicable only to applications where approximation is tolerated. Another drawback is that applications require multiple implementations to allow to dynamically switch between different levels of accuracy. PPEP [24] is an online prediction framework that, based on hardware performance events and on-chip temperature measurements, estimates the performance, power consumption and energy efficiency for each different CPU *P-state*. Therefore, it allows the definition of a power capping technique that can meet power targets in a single step without requiring any exploration. However, it does not consider the possibility of altering the number of CPU-core assigned to an application, thus it would provide sub-optimal performance for multi-thread applications with limited scalability.

3 PROBLEM STATEMENT AND PRELIMINARY ANALYSIS

As discussed, we consider the problem of maximizing the performance under a power cap for applications with multi-threaded workload that may show diverse scalability profiles. We consider two tuning parameters, the number of concurrent threads and the power state of the CPU cores. We also note that tuning the number of threads implies that the application is designed in such a way to permit such a change without damages to the correctness of its execution. In particular, we focus on CPU/memory bound applications based on the working thread pool paradigm which is adopted by wide-spread real-world applications such as commodity multi-threaded application servers or modern scientific computing platforms [15, 25].

We assume that the power state of CPU-cores can be changed via DVFS. We adhere to the ACPI standard notation, where *P0* identifies the CPU core state with maximum power and performance, and progressively *P1*, *P2*, ... identify states with less power and performance. When a CPU-core has no instructions to execute (e.g. when there is no candidate thread to be executed on that core), it can be transited from the full operating state, denoted with *C0*, to some of the available idle state, progressively denoted with *C1*, *C2*, ... When residing in one of these states, the core power consumption is highly reduced. Accordingly, when the number of running threads of the application is below the number of available CPU-cores, the transition of unused cores to some idle states is favored, thus reducing the overall power consumption.

To illustrate the effects on power consumption associated with the variation of *P-state* and the number of concurrent threads, we show in Figure 1 the results of an experiment where we run Intruder, which is one of the applications included in STAMP benchmark suite [2]. This suite offers various applications with multi-threaded workloads for performance analysis of in-memory transactional multiprocessing systems [22]. Intruder emulates a signature-based network intrusion detection system where network packets are processed in parallel by a tunable number of concurrent threads. We executed different runs of this application while changing *P-state* and the number of concurrent threads up to the number of available CPU-cores in the underlying machine. We used a machine with two Intel Xeon E5, 20 physical cores total, 256 ECC DDR4 memory, with core clock frequency ranging from 1.2 GHz (whose *P-state* is denoted as P11) to 2.2 GHz (denoted as P1), and TurboBoost from 2.2 GHz to 3.1 GHz (denoted as P0). Since we focus on the effects of the joint variation of the power state of CPU-cores and the thread-level parallelism, we consider power consumption relative to the CPU and memory subsystems, which we measured via the Intel RAPL interface [9]. The plot shows that the power consumption always increases while incrementing the number of concurrent threads or while decrementing *P-state*. We observed this behaviour also with all the other benchmark applications that we used in our study (as shown in Section 5). After all, this is not a surprising result. Indeed, adding more threads leads to keeping more CPU-cores in the operating state, thus increasing the overall power consumption, and decreasing *P-state* leads to use more power per core. Accordingly, we reasonably assume that this holds true with any workload.

We denote a system configuration with the couple (p, t), where p denotes the *P-state* and t is the number of threads. Given a power cap value C, if S is the set of all possible configurations, we denote as $S_{ac} \subseteq S$ the subset of all acceptable configurations, that is the configurations for which the power cap value is not violated. Denoting with $pwr(p, t)$ the power consumption with configuration (p, t), then $pwr(p, t) \leq C$ for each $(p, t) \in S_{ac}$. Based on our experimental observations, $pwr(p, t)$ is a monotonically increasing function with respect to both p and t. Thus, the subsets of acceptable and unacceptable configurations are separated by a frontier line. In Figure 2 we show an example of frontier line for the Intruder test case with $C = 50$ Watts.

Our goal is to find the configuration $(p, t)^* \in S_{ac}$ for which the performance of the application is maximized. Without loss of generality, we consider the application throughput as the performance metric. In any case, a different performance metric could be used, such as the response time. We denote as $thr(p, t)$ the application throughput with the configuration (p, t).

In Figure 3, we report the results of an experimental study we conducted to analyse the throughput curve $thr(p, t)$ while varying p and t. We used four multi-thread applications (still taken from STAMP), namely Intruder, Genome, Vacation and Ssca2. We selected these applications since they show very different behaviour in terms of scalability. Also, to carry out a more in-depth study on how contention and synchronization scenarios materialize, in our experiments we used two different implementations of each application. They are based on two different approaches to synchronize the accesse of threads to shared data. The first implementation uses a coarse-grained locking approach, where shared data accesses are

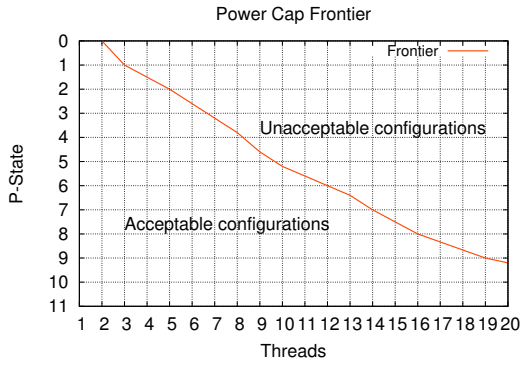

Figure 2: Frontier line between accepted and unaccepted configurations with C=50 watts

implemented as critical sections protected by a single global lock. The second one uses a fine-grained approach, relying on Software Transactional Memory [8], where shared data accesses are executed as (concurrent) transactions. This allowed us to analyze the scalability for antithetical synchronization techniques, spanning from pessimistic lock-based techniques to optimistic/speculative ones.

All plots in Figure 3 confirm our observations on the profile of the throughput curve. Indeed, in some cases, it shows an initial ascending part followed by a descending part. In other cases, the ascending part or the descending part do not exist. Also, the plots show that, when changing the application and/or the synchronization approach, the shape of the throughput curves may change. Particularly, the number of threads that provides the highest throughput may be different. In our experiments, it ranges from 1 (in the case of workloads with very limited scalability, such as for Intruder Lock-based, Vacation Lock-based and Ssca2 Lock-based) to 20 (in the case of scalable workloads as Genome Transaction-based or Vacation Transaction-based). Notably, in some cases it is in the middle (as for Intruder Transaction-based, Genome Lock-based or Ssca2 Transaction-based).

Another observation that comes out from the plots in Figure 3 is that, fixed the application and the synchronization approach, the throughput curves preserve the shape when varying p. The different curves appear translated, but the number of threads for which each curve reaches the maximum value (i.e. the highest throughput) does not change, unless for small and unpredictable variations generated by the measurement noise. Finally, the plots show that, keeping fixed the number of threads, the throughput value increases when decreasing p. We exploit these experimental findings to optimize our exploration-based technique that we presents in the next section.

4 THE ADAPTIVE POWER CAPPING TECHNIQUE

The adaptive power capping technique we propose is based on an on-line tuning strategy that periodically performs an exploration procedure. This procedure finds the optimal configuration $(p, t)^*$, which is actuated and kept until the exploration procedure restarts

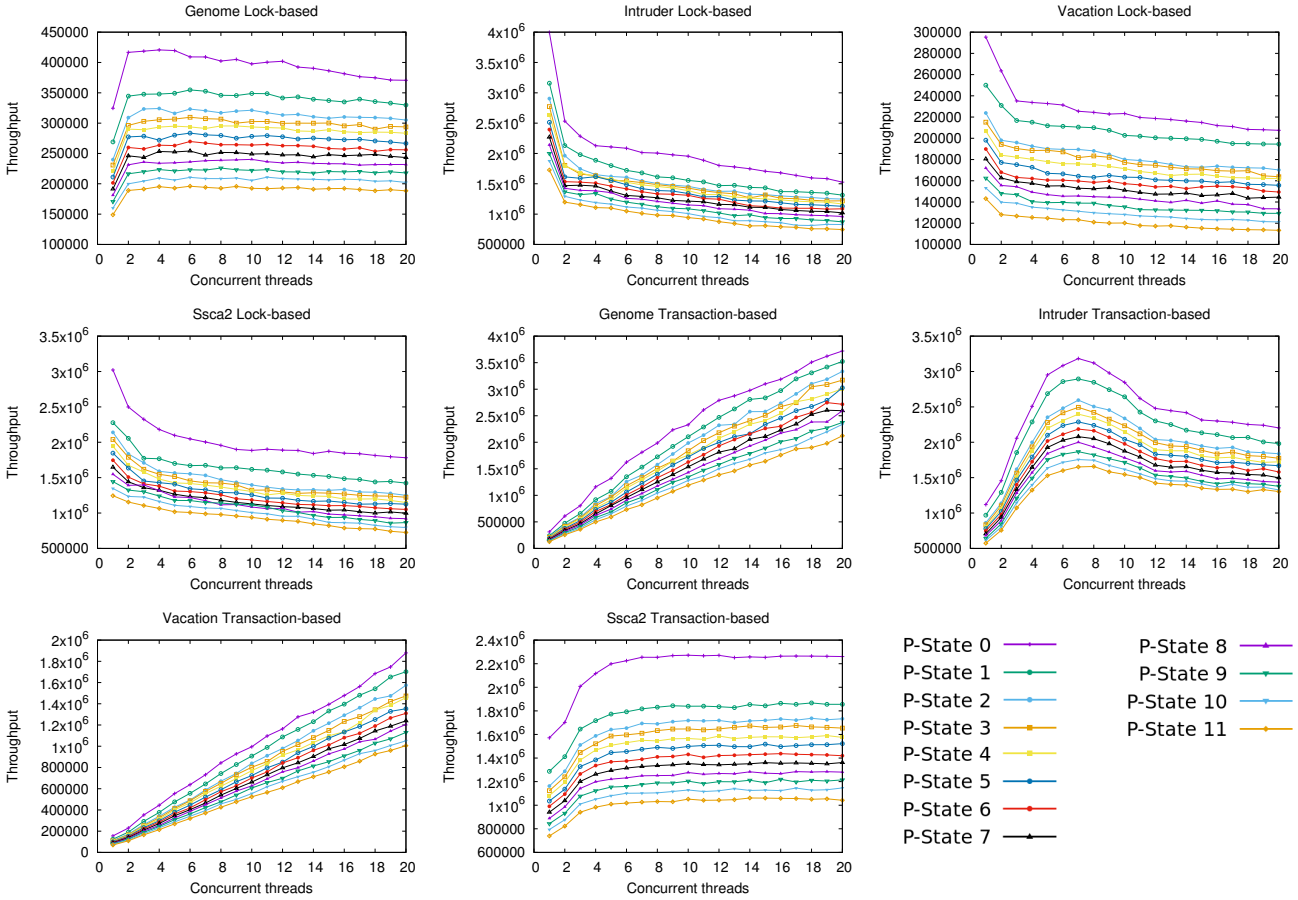

Figure 3: Throughput vs. number of concurrent threads

after a pre-established period. This allows to capture possible variations of the workload profile along time, which may lead to a different optimal configuration.

In short, the exploration procedure records the measures of the power consumption and the throughput of the application while moving along configurations within a specific path. At the end of the exploration, the one with the highest throughput is selected. The procedure is able to identify the optimal configuration by exploring only a subset of all possible configurations. We note that the full set of configurations may be very large, particularly when a large number of CPU-cores and/or *p-states* are available. Thus, reducing the exploration space is fundamental for an on-line exploration-based strategy.

4.1 The Exploration Procedure

The exploration procedure takes as input a starting configuration (p^s, t^s) and a power cap value C and returns $(p, t)^*$. For the first execution of the procedure, the starting configuration can be arbitrarily selected, while in subsequent executions it starts from the output configuration of the previous one. We note that the shapes of the throughput curves and the observations that we made in our preliminary analysis allow to exclude some configurations from the exploration, thus reducing the configuration exploration space. Specifically, if during the exploration:

(1) a configuration (p^j, t^k) such that $thr(p^j, t^k) \leq thr(p^j, t^k - 1)$ is found then all configurations (p, t) where $t \geq t^k$, for whichever p, can be excluded (since we are in the descending part of the throughput curve and since the throughput curves preserve the shape while varying *P-state*).
(2) a configuration (p^j, t^k) such that $pwr(p^j, t^k) \leq C$ is found then all configurations (p, t^k) with $p > p^k$ can be excluded (since increasing *P-state* reduces the application throughput).
(3) a configuration (p^j, t^k) such that $pwr(p^j, t^k) > C$ is found then all configurations (p, t) where $t \geq t^k$ and $p \leq p^k$ can be excluded (since decreasing *P-state* or increasing the number of concurrent threads increments the power consumption).

Based on the above observations, we built an exploration procedure articulated in 3 phases, plus a final selection phase. The phases are described below. To help the reader while reading the description, a graphical example is shown in Figure 4, which refers to a test case where the number of concurrent threads providing the highest throughput is equal to 15.

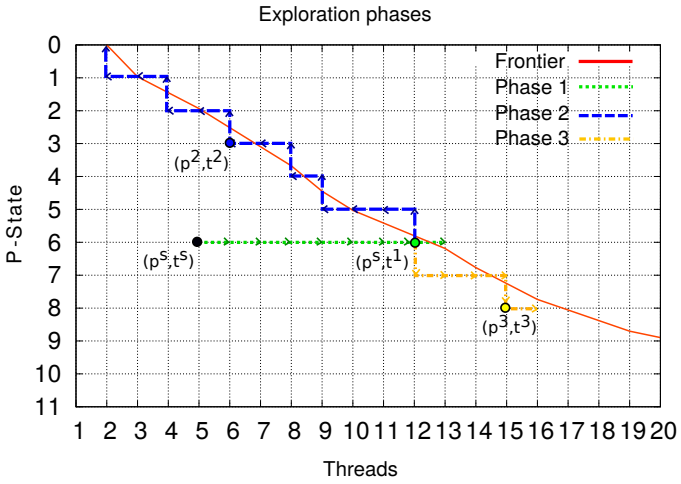

Figure 4: Example of exploration phases performed by the basic strategy

Phase 1. This phase starts from the initial configuration (p^s, t^s), and aims at finding, keeping fixed p^s, the number of threads providing the highest throughput without violating the power cap. We denote as (p^s, t^1) the configuration returned by this phase. It performs a search inspired by the hill-climbing technique [23]. Specifically, it starts incrementing by one the number of threads, and continues if the throughput increases and the power cap is not violated (if the throughput increases it means that it is moving along the ascending part of the throughput curve). It stops when the throughput starts decreasing, when the power cap is violated or when the maximum number of threads (which optionally can be pre-established by the user) has been reached. Finally, it returns the configuration for which it measured the highest throughput and which is within the power cap. If the throughput does not grow after the first increment, or the power cap is violated, it starts decreasing the number of threads (since it is moving along the descending part of the throughput curve, or the power consumption must be reduced) until the throughput starts decreasing. Then, it returns the configuration with the highest throughput if it does not violate the power cap. Otherwise, if all the explored configurations violate the power cap, or if the exploration reaches a number of threads equal to 1, it returns $(p^s, 1)$. In the example in Figure 4, the exploration during phase 1 is represented by the green line. It starts with $(p^s, t^s) = (6, 5)$, then increases the number of threads and terminates when it reaches configuration $(6, 13)$ since it violates the power cap. It returns $(p^s, t^1) = (6, 12)$, which is within the power cap.

Phase 2. This phase starts from the configuration (p^s, t^1) returned by phase 1 and is executed only if this configuration 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 increasing the value of *P-state* leads to both higher core performance and higher power consumption). Specifically, it moves from the current configuration (p, t) to configuration $(p - 1, t)$. If the latter configuration does not violate the power cap, it continues to reduce the value of *P-state*. If a configuration such that $pwr(p, t) > C$ is reached, it starts reducing the number of threads, thus moving to configuration $(p, t-1)$, then $(p, t-2)$ and so on (since decreasing the number of concurrent threads reduces the power consumption) until the power cap is not violated. After, it restarts the exploration by decreasing the value of *P-state*. The exploration terminates when p reaches 0 and the current configuration does not violate the power cap, when it reaches configuration $(0, 1)$, or when a configuration with $t = 1$ violates the power cap. Then, among the explored configurations, Phase 2 returns the configuration (that we denote as (p^2, t^2)) with the highest throughput within the power cap, or $(0, 1)$ if none of the explored configurations is within the power cap. In Figure 4, the exploration of Phase 2 is shown by the blue line. It starts from $(p^s, t^1) = (6, 12)$, and then explores up to configuration $(0, 1)$. It returns $(p^2, t^2) = (3, 6)$.

Phase 3. This phase starts again from the configuration returned by Phase 1, i.e. (p^s, t^1), and aims at continuing the exploration for higher values of *P-state*. If the configuration returned by Phase 1 is such that t^1 is the number of threads providing the highest throughput and is within the power cap, Phase 3 is not executed (since decrementing the value of *P-state* leads to lower throughput). If not, it increments by one the value of *P-state* and starts increasing the number of concurrent threads 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, it increments by one the value of *P-state* and starts again incrementing the number of threads. In all the other cases the exploration terminates. Then, phase 3 returns the explored configuration (that we denote as (p^3, t^3)) with the highest throughput within the power cap, or it returns (p_{max}, t^1) (where p_{max} is the maximum value of *P-state*) if all the explored configurations are within the power cap. In Figure 4, the exploration of Phase 3 is represented by the yellow line. It starts from $(p^s, t^1) = (6, 12)$, then explores up to configuration $(8, 16)$, where it stops since the throughput decreases (we remark that in the example the number of concurrent threads providing the highest throughput is equal to 15). It returns $(p^3, t^3) = (8, 15)$.

Final phase: this phase selects the configuration with the highest throughput between the configurations (p^s, t^1), (p^2, t^2) and (p^3, t^3), which does not violate the power cap, or returns *null* if none of them is within the power cap.

4.2 Proof of Optimality

In this section we prove that the proposed exploration procedure finds the optimal configuration in the bi-dimensional space of configurations defined by all combinations of active threads and CPU *P-state*. We note that solving this problem in linear time is not trivial, since the approach of finding the optimal solution for each dimension independently—which might be trivial with some hill-climbing approach under Assumption 1—does not compose to the bi-dimensional optimum. We initially present the set of assumptions our proof relies on. We recall that all these assumptions originate from the experimental results we discussed in Section 3.

Assumption 1. Fixed *P-state* and increasing t from 0 to t_{max}, the throughput curve behaves as follows:

(1) initially increases, reaches its maximum value, then decreases, otherwise
(2) monotonically increases, otherwise

(3) monotonically decreases.

Assumption 2. If $thr(p^j, t^k) > thr(p^j, t^k + 1)$ then for each p we have $thr(p, t^k) > thr(p, t^k + 1)$. Also, if $thr(p^j, t^k) > thr(p^j, t^k - 1)$ then for each p we have $thr(p, t^k) > thr(p, t^k - 1)$. In other words, if for some *P-state* and t^k threads the throughput decreases (increases) when adding (removing) one thread, then this holds true for whichever *P-state*. Overall, the ordering relations on the throughput values when changing the number of threads are not effected by *P-state*.

Assumption 3. if $p^j < p^k$ then $thr(p^j, t) > thr(p^k, t)$ for whichever t. In other words, when decreasing the value of *P-state* the throughput always increases for whichever number of threads;

Assumption 4. If $p^j < p^k$ then $pwr(p^j, t) > pwr(p^k, t)$, and if $t^j > t^k$ then $pwr(p, t^j) > pwr(p, t^k)$. In other words, the power consumption increases when decreasing *P-state* or when increasing the number of threads.

STATEMENT. The exploration procedure presented in Section 4.1 is guaranteed to find the optimal configuration of CPU *P-state* and thread-level parallelism.

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

We show that Phases 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 by Phase 1. Phase 1 explores configurations within S1. Specifically, it keeps fixed p^s and explores while varying only the number of threads t. Phase 1 uses the hill-climbing search. By Assumption 1, the function $thr(p^s, t)$ has only one local maximum, thus it corresponds to the global maximum. Accordingly, the hill-climbing search trivially can find the maximum [23], 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 threads which is within the power cap is found. Also in this case, it is the optimal configuration in S_1.

Outcome by Phase 2. We recall that Phase 2 starts exploring from the configuration returned by Phase 1, denoted as (p^s, t^1), which is the optimal one with *P-state* equal to p^s, unless none of the configurations with *P-state* equal to p^s is within the power cap. In the latter case, Phase 1 returns $(p^s, 1)$. Also, we recall that Phase 2 explores moving towards lower *P-states* and a lower number of threads. For Assumption 2, the number of threads that provides the maximum throughput does not change when decreasing *P-state*. Accordingly, if (p^s, t^1) is the optimal configuration fixed p^s, then the optimal configuration for the sub-space S_2 must have a number of threads less than or equal to t^1. Specifically, if $pwr(p^s - 1, t^1) < C$ then the optimal configuration with *P-state* equal to $p^s - 1$ has still t^1 threads. Otherwise, if the power cap is violated, the number of threads has to be reduced to stay within the power cap. Also, this mean that reducing the number of threads leads to reduce the throughput, since the above situation can arise only if we are in the ascending part of the throughput curve. Accordingly, in this case the optimal configuration is the first one that is within the power cap while reducing the threads. Phase 2 follows exactly this behaviour, i.e. it first moves to *P-state* equal to $p^s - 1$, and if $pwr(p^s - 1, t^1) > C$ then it reduces the number of threads 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 performs this search for each *P-state* such that $p \in [0, p^s - 1]$. Thus, it finds the optimal configuration for each *P-state* in the sub-space S_2. Finally, it selects the optimal one of them, thus finding the optimal configuration in the sub-space S_2.

Outcome by Phase 3. We remark that Phase 3 starts exploring from the configuration returned by Phase 1, and explores moving towards higher *P-states* and a higher number of threads. Also, we remark that Phase 3 is not executed if the configuration returned by Phase 1 is such that t^1 is the number of threads that provides the highest throughput and is within the power cap. Indeed, in this case, the throughput for any configuration with a number of threads higher than t^1 and any higher *P-state* is lower for Assumption 2. Hence, Phase 3 is executed only if the number of threads that provides the highest throughput is higher than t^1, but it violates the power cap with *P-state* equal to p^1. This means that t^1 is along the ascending part of the throughput curve due to Assumption 1. Also, this holds true for any *P-state* higher than p^1 for Assumption 2. Accordingly, the throughput with any configuration in the sub-space S_3 with a number of threads less than t^1 is lower. Consequently, the optimal configuration for *P-state* equal to $p^s + 1$ must have a number of threads higher than t^1. Phase 3 first moves to *P-state* equal to $p^s + 1$, then it starts increasing the number of threads and stops when the power cap is violated or the throughput decreases. Accordingly, it finds the optimal configuration for *P-state* equal to $p^s + 1$. After, Phase 3 performs this search 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. Finally, it selects the optimal one of them, thus finding the optimal configuration in the sub-space S_3. □

4.3 Time Complexity Analysis

We estimate the time complexity of the exploration procedure as the number of exploration steps required to return the optimal configuration. We evaluate the time complexity of each exploration phase separately:

- **Phase 1.** Each configuration with a different number of concurrent threads and $p = p^s$ is explored at most once, thus the time complexity is $O(t_{max})$;
- **Phase 2.** Starting from a configuration (p, t), Phase 2 either reduces the value of p or reduces t. Starting from the configuration returned by Phase 1, it can reduce p at most p_{max} times, and can reduce t at most t_{max} times. Thus, the time complexity of Phase 2 is $O(p_{max} + t_{max})$;
- **Phase 3.** Starting from a configuration (p, t), Phase 3 either increments the value of p or increments t. Thus, for the same

reasoning used in Phase 2, the time complexity of Phase 3 is $O(p_{max} + t_{max})$.

Therefore, the overall time complexity of the exploration procedure is $O(p_{max} + t_{max})$.

4.4 The Enhanced Tuning Strategy

In this section, we present an improvement of our tuning strategy that allows to further improve performance and reduce the probability to violate the power cap. It takes advantage of two practical factors:

(1) The power consumption with the optimal configuration $(p,t)^*$ may be lower than the power cap, thus $C - pwr(p,t)^* > 0$. This is a consequence of the discrete set of power consumption values resulting from the discrete domain of *P-states*. Statistically, the greater the difference of power consumption between adjacent configurations, the larger the difference between C and $pwr(p,t)^*$.

(2) The power consumption with the optimal configuration $(p,t)^*$ may change along the time interval in-between two subsequent exploration procedures due to possible variations of the workload profile. Thus, $pwr(p,t)^*$ could increase over the power cap. Similarly, the application power profile might change such that some configuration with both higher performance and power consumption than $(p,t)^*$ might enter the set of acceptable configurations.

Our enhanced tuning strategy performs fluctuations between different configurations to mimic a continuous domain of power consumption values, while also accounting for possible variations of the application power profile. It relies on the same exploration technique used by the basic strategy—without introducing any further step in the exploration—but in addition to $pwr(p,t)^*$ it also selects two other configurations:

- $(p,t)^H$ the explored configuration with highest throughput such that $pwr(p,t)^H < C * (1+h)$, and
- $(p,t)^L$ the explored configuration with highest throughput such that $pwr(p,t)^L < C * (1-h)$.

The parameter h defines the ideal distance between the power consumption of $(p,t)^H$ and $(p,t)^L$ with respect to the power cap. We note that $thr(p,t)^L \leq thr(p,t)^* \leq thr(p,t)^H$. In the time interval between the end of the exploration procedure and the start of the next one, the enhanced strategy performs fluctuations between $(p,t)^H$, $pwr(p,t)^*$ and $pwr(p,t)^L$ to maximize the performance over a window w, while keeping the average power consumption along the window lower than C. When $pwr(p,t)^* < C$, the strategy moves between $(p,t)^*$ and $(p,t)^H$, selecting the former whenever the average power consumption along the window is above the power cap, while selecting the latter in the opposite case. The result of this fluctuation can provide a performance increase over $thr(p,t)^*$. Differently, if $pwr(p,t)^* > C$ the variation of configurations is performed between $(p,t)^*$ and $(p,t)^L$, using the latter to reduce the average power consumption over the window. In this scenario, the enhanced strategy can provide a reduction in the power cap violation compared to the static exploitation of $(p,t)^*$. To limit the fluctuation frequency, an upper and a lower tolerance threshold like $C + l$ and $C - l$ can be used. Temporary power cap violations of a few milliseconds are not relevant as power consumption is generally computed as an average value at the granularity of seconds.

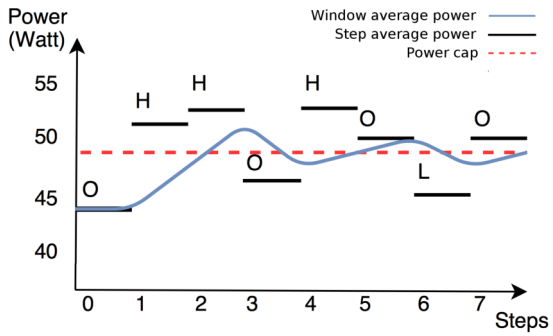

Figure 5: Example of fluctuations of configurations performed by the enhanced strategy in-between different exploration procedures. Window size is set to 8. O denotes the configuration $(p,t)^*$, H the configuration $(p,t)^H$ and L the configuration $(p,t)^L$. These configurations are fixed along each window but their power consumption could change.

To adapt to workload variations, at the end of each window, if $pwr(p,t)^L > C$ the *P-state* of $(p,t)^*$ is shifted up by one in order to reduce its power consumption. Moreover, $(p,t)^H$ and $(p,t)^L$ are also set to different configurations such that they have the same number of threads as $(p,t)^*$ but with *P-state* decremented or incremented by one respectively. This lowers the overall power consumption of the configurations, thus allowing to promptly adapt to the increase in the application power profile. Otherwise, if $pwr(p,t)^H < C$, the same modifications are applied except that the *P-state* of $(p,t)^*$ is shifted down by one instead of up. The possibility of increasing the power consumption of the configurations creates the opportunity for further performance gains. In both situations, we only modify the *P-state* and set for all configurations the number of threads equal to the configuration $(p,t)^*$ as modifying the *P-state* always provides either an increase or a decrease in both performance and power consumption while changing the number of threads might provide different performance results based on the workload characteristics.

A pseudo-code representation of the algorithm implemented by the enhanced strategy is presented below.

Figure 5 shows the fluctuations performed by the enhanced strategy along a window with $w = 8$. From step 0 to step 4 the strategy fluctuates between $(p,t)^*$ and $(p,t)^H$, allowing increased performance compared to the basic strategy. In step 5, to decrease the average power consumption along the window, the configuration $(p,t)^*$ is selected. However, its power consumption has increased since its last exploitation and has become higher than the power cap. In step 6, the enhanced strategy selects $(p,t)^L$ to reduce power cap violations compared to the basic strategy which would have been static to configuration $(p,t)^*$ until the next exploration procedure. Configuration $(p,t)^*$ is selected in step 7 to conclude the window. Despite no configuration shows a power consumption similar to the power cap, the average power during the window converges to its value.

Algorithm 1 Fluctuation algorithm used in the enhanced strategy

```
procedure FLUCTUATE
Require: Power cap C
Require: Configuration O, T and H with attributes (pstate, threads, power)
Require: Power cap threshold l
Require: Window size w
    step ← 0
    windowPower ← 0
    windowTime ← 0
    selectedConfig ← O
    while step < w do
        currentPower, time ← getMeasurements(selectedConfig)
        set power of the measured configuration to currentPower
        windowPower ← (windowPower ∗ windowTime+
        currentPower ∗ time)/(time + currentTime)
        windowTime ← windowTime + time
        slot ← slot + 1
        if windowPower < C ∗ (1 − l) then
            selectedConfig ← H
        else if windowPower > C ∗ (1 + l) then
            if O.power < C then
                selectedConfig ← O
            else
                selectedConfig ← L
        else ▷ window power consumption close to power cap
            if H.power < C then
                selectedConfig ← H
            else if O.power < C then
                selectedConfig ← O
            else
                selectedConfig ← L
    if H.power < C or L.power > C then
        if H.power < C then ▷ can increase the power consumption
            O.pstate ← O.pstate − 1
        else ▷ should reduce power consumption
            O.pstate ← O.pstate + 1
        H.threads ← O.threads
        L.threads ← O.threads
        H.pstate ← O.pstate − 1
        L.pstate ← O.pstate + 1
```

5 EXPERIMENTAL RESULTS

In this section, we present the results of an experimental study we conducted to assess the proposed power capping technique. As in previous studies on power capping (e.g. [12, 19]), we consider two evaluation metrics, the application performance and the average power cap error. The latter is the average difference between the power consumption and the power cap value along time intervals where the power cap is violated. When assessing performance and power cap errors, we also include measurements gathered along the exploration procedure. We run experiments for all application scenarios that we considered in our preliminary study (see Section 3). Thus we use Intruder, Genome,Vacation and Ssca2 as benchmark applications from STAMP, with either locks or transactions as the synchronization method. As hinted, these applications (and the different instances of the synchronization support) were specifically selected to cover a wide range of different scalability scenarios. We compared our technique with:

(1) a reference power capping technique, referred to as baseline, that selects the configuration with the lowest *P-state* from the set of configurations such that the number of threads is the highest among the configurations with power consumption lower than the power cap. It implements the selection strategy proposed in [19];
(2) a technique, referred to as dual-phase, that initially tunes the number of threads starting from the lowest *P-state*, and subsequently tunes the CPU *P-state* keeping the number of threads fixed. The initial phase is equivalent to phase 1 of the proposed exploration procedure. The selection strategy of this technique is similar to the one presented in [26].

The comparison of our proposal with the technique in point (1) allows to quantify the performance benefits achievable by properly allocating the power budget taking into account the scalability level of the specific multi-threaded workload. Additionally, the inclusion of the dual-phase technique listed in point (2) in the evaluation allows quantifying the possible performance benefits achievable by exploring the whole bi-dimensional space of configurations—as we do in our approach—over two distinct mono-dimensional explorations, which might not find the optimal configuration. We should note that, despite exploring a larger set of configurations, the technique we propose has the same time complexity of the dual-phase technique.

5.1 Implementation Details

We developed a controller module that implements our technique and the baseline technique.[1] All software of our experimental study, including benchmark applications, is developed in C language for Linux. The controller module alters the number of concurrent threads exploiting the *pause()* system call and thread-specific signals for reactivation. The CPU *P-state* is regulated through the *cpufreq* Linux sub-system, while energy readings are obtained from the *powercap* sub-system. Both these sub-systems are included by default in recent versions of the Linux kernel and expose their respective interface through the */sys* virtual file system.

At each step, the exploration procedure relies on statistical results of a previous step, such as average power consumption and throughput, to define the subsequent configuration to explore. Each step of statistics collection is determined by a fixed amount of units of work processed. We cannot rely on application independent metrics, such as the number of CPU retired operation, since it would also consider instructions related to spin-locking or aborted transactions that do not provide execution progress. For applications based on locks we defined the unit of work as the execution of one critical section guarded by a global lock. Differently, for transactions we define the unit of work as one commit. The statistics are collected in a round-robin fashion by all the active threads to reduce execution overhead and provide NUMA-aware results in modern multi-package systems.

[1] See `github.com/HPDCS/EPADS`

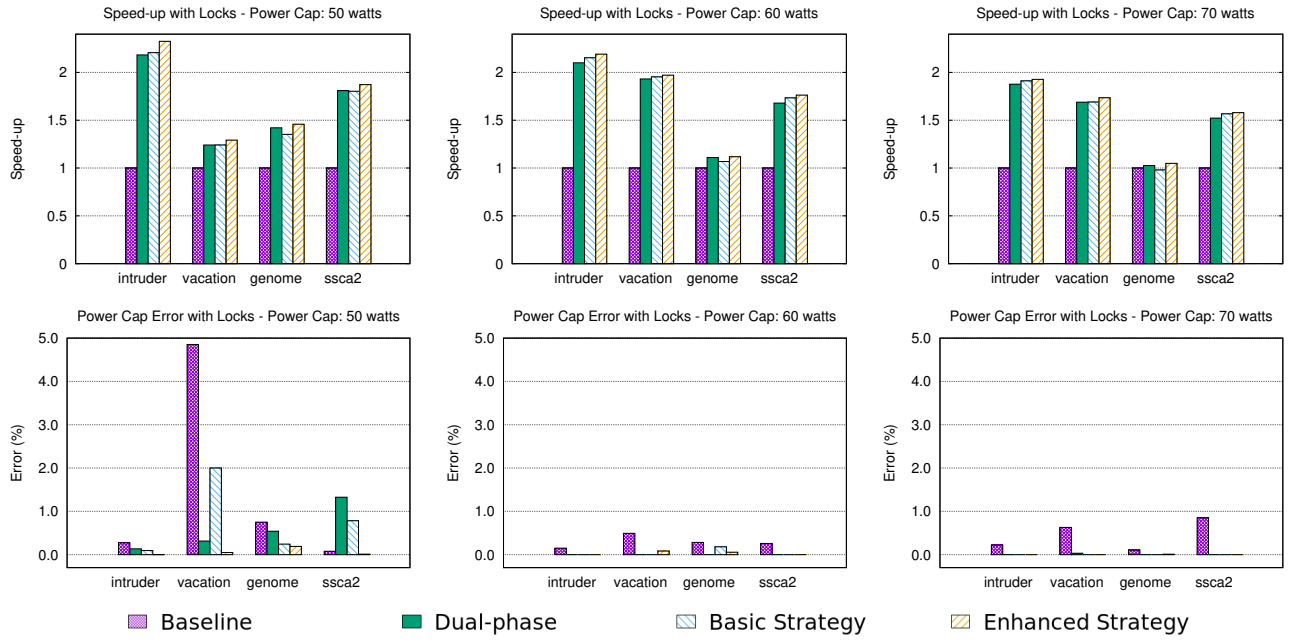

Figure 6: Speed-up and power cap error with locks

For the executions presented in the experimental results, we set the units of work per step to 5000, resulting in tens of milliseconds per step for all the considered applications and synchronization methods. In addition, we set to 150 the number of steps required to restart the exploration procedure after the conclusion of the previous exploration. Regarding the parameters of the enhanced strategy, we set the window size (w) to 10, the maximum power consumption for $(p,t)^H$ and $(p,t)^L$ to respectively the 10% higher or lower than the power cap(h) and the fluctuation threshold (l) to 1%. We always used the same parameters for all applications and synchronization techniques. Autonomic tuning of these parameters at run-time—possibly leading to increased performance benefits—will be explored in a future work. With the tested parameters, the overheads of changing configuration—dominated by *P-state* switching—and the cost of performance and power measurements are lower than 2% for all the considered executions.

5.2 Experimental Results

We consider both the tuning strategies of our technique referred to as basic strategy and enhanced strategy. We analyze the performance results of our strategies in terms of speed-up with respect to the throughput of the baseline technique. As anticipated, we also compare the average power cap error. For each test case, we present the results with three different power cap values, i.e. 50, 60 and 70 watts.

Results for the case of lock-based synchronization are reported in Figure 6. Overall, the results show an evident performance improvement with both strategies of our technique with respect to the baseline technique. Only for the case of Genome the performance is comparable. In the best cases, i.e. with Intruder, the performance improvement reaches 2.2x (2.32x) and 2.15x (2.19x) for the basic (enhanced) strategy when the power cap is equal to 50 and 60 watts respectively, and it is close to 1.9x for both the proposed strategies with power cap set to 70 watts. The enhanced strategy further improves performance compared to the baseline technique by up to 12.5% in Intruder at 50 watts, and by 5.3% on average. For lock-based synchronization, the results of the dual-phase technique are similar to those achieved by the baseline technique.

As for the power cap error, with both the strategies of our technique and the dual-phase technique, it is clearly reduced compared to the baseline. Also, the results show that with the enhanced strategy in many cases there is a reduction of the power cap error compared to the basic strategy. Indeed, except for the case of Vacation with power cap set to 60 watts, where it is increased by less than 0.1%, the error with the enhanced strategy is lower. In the best case it is about 0.1%, while it is about 2% and 4.8% with the basic strategy and the baseline technique, respectively.

Results for the case of transaction-based synchronization are reported in Figure 7. Overall, the performance results confirm the advantage of our technique compared to the baseline technique. However, with transactions the speed-up is generally slightly lower than with locks. In the best cases, it reaches about 1.9x. Also, there is one case (with Genome and power cap = 50 watts) where it is slightly less that 1 with both the strategies. As for the power cap error, it increases with the basic strategy compared to the case with locks, overcoming the error of the baseline technique in most of the cases. However, it does not overcome 2% in all cases. The error is considerably reduced with the enhanced strategy. Particularly, it is clearly lower than the baseline technique with all applications when the power cap is equal to 50 watts and with Intruder when the power cap is equals to 60 watts, while the results are similar for the other power cap values. In addition, the enhanced strategy can

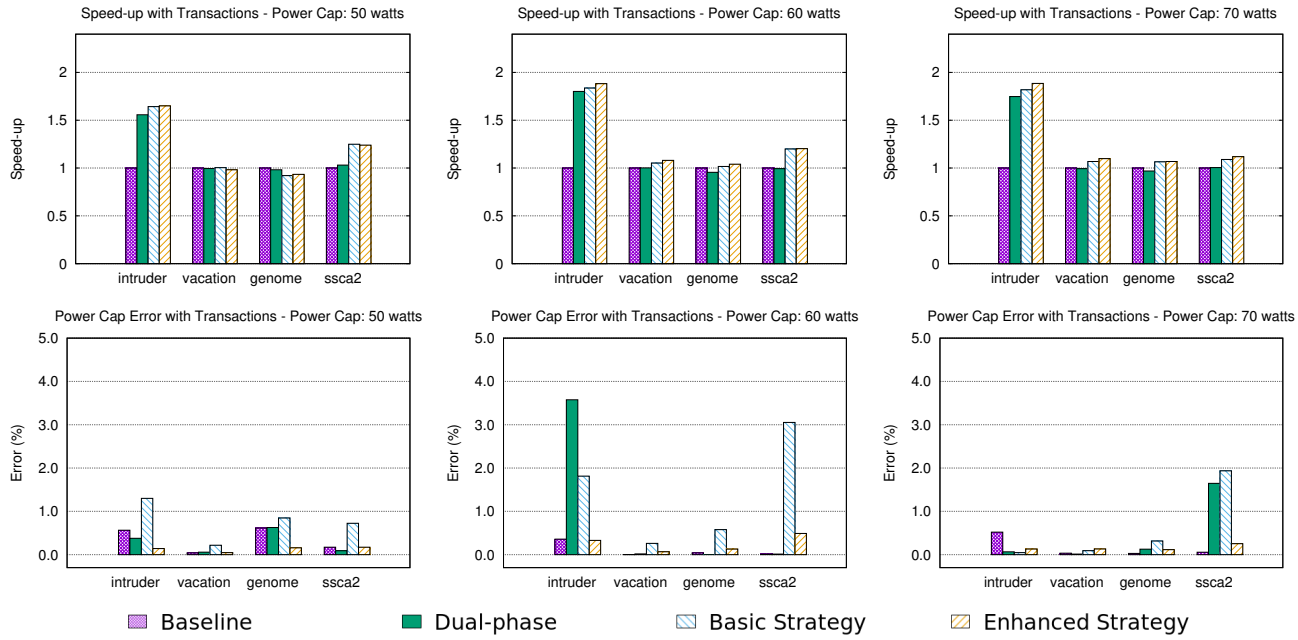

Figure 7: Speed-up and power cap error with transactions

further increase performance by up to to 8% (Vacation with power cap set to 50 watts) and by 3.5% on average. Differently from the lock-based case, both strategies of the proposed technique show an higher speed-up compared to the dual-phase technique by up to 21% (ssca2 with power cap set to 50), and by 7.7% and 10.7% on average for the basic strategy and the enhanced strategy respectively.

5.3 Analysis of the Results

As a first observation, the results show that in various cases with locks, the error of our technique and of the dual-phase technique is very close to zero. This is due to the fact that, in our study, the scalability is limited for all applications when using locks. In these scenarios, the number of concurrent threads providing higher throughput (that is selected by our technique and by the dual-phase technique) is low, thus the value of *P-state* can be changed up to 0 while the power cap frontier is still far. This keeps the error very close to 0 since it is unlikely that the power cap is violated during the exploration procedure or due to workload variations.

The error is generally reduced with the enhanced strategy compared to the basic strategy, while also improving performance. This arises since the former is able to react along the time between two consecutive exploration procedures to the possible variations of the power consumption of the selected configurations, as discussed at the end of Section 4.4.

The speed-up with our technique is less than 1 only in one case, i.e. for Genome with transactions when the power cap value is equal to 50 watts. We note that Genome with transactions is highly scalable (see Figure 3). This leads both the baseline technique and our technique to select 20 as the number of concurrent threads. As shown by the plot in Figure 3, the throughput of Genome with transactions is subject to noise when close to 20 threads . Also, we remark that our technique is able to react to workload variations also in terms of scalability. In this scenario, these factors cause lower performance with our technique due to noise, which sometimes (wrongly) leads to temporarily selecting a less than optimal number of concurrent threads.

As expected, for lock-based synchronization the proposed technique shows similar results to the dual-phase technique since both techniques return the same configuration when the ascending part of the throughput curve is missing. For transaction-based synchronization, the best speed-up improvements over the dual-phase technique are obtained for Ssca2 and Genome which show a less than linear ascending part of the throughput curve for each fixed *P-state* (Figure 3). As the most significant example, in Ssca2 the throughput slightly increases when increasing the number of threads from 6 to 15 which makes the dual-phase technique select a configuration with 15 threads. Differently, the proposed technique allocates the power budget more efficiently by selecting a configuration with a lower number of threads at an increased frequency. We should note that the benefits of the proposed technique over the dual-phase technique are not limited to applications that rely on transactional-based synchronization. Effectively, performance benefits should be obtained for any application with a throughput function that shows an ascending part followed by a descending one, or only an ascending part that is less than linear.

Overall, the results of our experiments study show that it is possible to achieve significant performance benefits by appropriately selecting the number of concurrent threads and CPU *P-state* taking into consideration the scalability of the specific multi-threaded workload. As expected, compared to the baseline technique, the proposed solutions achieve the best results with poorly scalable applications, i.e. where contention is not minimal. Compared to the

dual-phase technique, the exploration of the whole bi-dimensional space of configurations performed by the proposed technique can provide an appreciable improvement in performance for some applications, while achieving the same results for others. Finally, the enhanced strategy manages to further improve performance and reduce the power cap error over the basic strategy.

6 CONCLUSIONS

In this paper, we have proposed a novel power capping technique for dynamic tuning the number of concurrent threads and core power states for the case of multi-thread applications that materialize diverse scalability levels, also depending on the specific support for synchronizing the accesses to shared data. The technique is able to find the optimal configuration in linear time, ensuring the maximum performance achievable within the power cap. We also present an improvement of the technique that induces fluctuations between different configurations to efficiently exploit the full power budget, resulting in both increased performance and reduced power cap errors. We have shown that, compared to the baseline technique, our strategy provides an average speed-up of 1.48x, with individual test cases reaching up to 2.32x. Furthermore, we have shown that, by exploring the overall bi-dimensional space of configurations, the proposed technique can improve performance by up to 21% compared to techniques that tune the number of threads and the CPU performance state independently.

REFERENCES

[1] Md Abdullah Shahneous Bari, Nicholas Chaimov, Abid M. Malik, Kevin A. Huck, Barbara Chapman, Allen D. Malony, and Osman Sarood. 2016. ARCS: Adaptive runtime configuration selection for power-constrained OpenMP applications. *Proceedings - IEEE International Conference on Cluster Computing, ICCC* (2016), 461–470. https://doi.org/10.1109/CLUSTER.2016.39

[2] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. *4th International Symposium on Workload Characterization*, 35–46. https://doi.org/10.1109/IISWC.2008.4636089

[3] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R Hanebutte, and Onur Mutlu. 2011. Memory Power Management via Dynamic Voltage/Frequency Scaling. *Proceedings of the 8th ACM International Conference on Autonomic Computing* (2011), 31–40. https://doi.org/10.1145/1998582.1998590

[4] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory power estimation and capping. *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on* (2010), 189–194. https://doi.org/10.1145/1840845.1840883

[5] Qingyuan Deng, Luiz Ramos, Ricardo Bianchini, David Meisner, and Thomas Wenisch. 2012. Active low-power modes for main memory with memScale. *IEEE Micro* 32, 3 (2012), 60–69. https://doi.org/10.1109/MM.2012.21

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

[7] Neha Gholkar, Frank Mueller, and Barry Rountree. 2016. Power Tuning HPC Jobs on Power-Constrained Systems. *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation - PACT '16* (2016), 179–191. https://doi.org/10.1145/2967938.2967961

[8] Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory, 2nd Edition* (2nd ed.). Morgan and Claypool Publishers.

[9] Intel. 2011. Intel 64 and IA-32 Architectures Software Developer Manual, Volume 3C: System Programming Guide, Part 3. (2011).

[10] Canturk Isci, Alper Buyuktosunoglu, Chen Yong Cher, Pradip Bose, and Margaret Martonosi. 2006. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO* (2006), 347–358. https://doi.org/10.1109/MICRO.2006.8

[11] Anil Kanduri, Mohammad-Hashem Haghbayan, Amir M. Rahmani, Pasi Liljeberg, Axel Jantsch, Nikil Dutt, and Hannu Tenhunen. 2016. Approximation knob: power capping meets energy efficiency. *Proceedings of the 35th International Conference on Computer-Aided Design - ICCAD '16* (2016), 1–8. https://doi.org/10.1145/2966986.2967002

[12] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. 2008. Power Capping: A Prelude to Power Shifting. *Cluster Computing* 11, 2 (June 2008), 183–195. https://doi.org/10.1007/s10586-007-0045-4

[13] Yanpei Liu, Guilherme Cox, Qingyuan Deng, Stark C. Draper, and Ricardo Bianchini. 2016. FastCap: An efficient and fair algorithm for power capping in many-core systems. *ISPASS 2016 - International Symposium on Performance Analysis of Systems and Software* 3 (2016), 57–68. https://doi.org/10.1109/ISPASS.2016.7482074

[14] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. 2013. Hierarchical power management for asymmetric multi-core in dark silicon era. *Proceedings of the 50th Annual Design Automation Conference on - DAC '13* (2013), 1. https://doi.org/10.1145/2463209.2488949

[15] Oracle. 2017. Plug into the Cloud with Oracle Database 12C (wite paper). http://www.oracle.com/technetwork/database/plug-into-cloud-wp-12c-1896100.pdf. (2017).

[16] Venkatesh Pallipadi and Alexey Starikovskiy. 2006. The ondemand governor: past, present and future. In *Proceedings of Linux Symposium, vol. 2, pp. 223-238.* https://doi.org/pub/linux/kernel/people/lenb/acpi/doc/OLS2006-ondemand-paper.pdf

[17] Allan K. Porterfield, Stephen L. Olivier, Sridutt Bhalachandra, and Jan F. Prins. 2013. 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. https://doi.org/10.1109/IPDPSW.2013.15

[18] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. 2008. No âĂIJ Power âĂİ Struggles : Coordinated Multi-level Power Management for the Data Center. *Solutions* 36 (2008), 48–59. https://doi.org/10.1145/1346281.1346289

[19] Sherief Reda, Ryan Cochran, and Ayse Coskun. 2012. Adaptive Power Capping for Servers with Multithreaded Workloads. *IEEE Micro* 32, 5 (Sept. 2012), 64–75. https://doi.org/10.1109/MM.2012.59

[20] D. Rughetti, P. Di Sanzo, and A. Pellegrini. 2014. Adaptive Transactional Memories: Performance and Energy Consumption Tradeoffs. In *Network Cloud Computing and Applications (NCCA), 2014 IEEE 3rd Symposium on*. IEEE Computer Society, 105–112. https://doi.org/10.1109/NCCA.2014.25

[21] Osman Sarood, Akhil Langer, Laxmikant Kale, Barry Rountree, and Bronis De Supinski. 2013. Optimizing power allocation to CPU and memory subsystems in overprovisioned HPC systems. In *Proceedings - IEEE International Conference on Cluster Computing, ICCC*. https://doi.org/10.1109/CLUSTER.2013.6702684

[22] Nir Shavit and Dan Touitou. 1995. Software transactional memory. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*. ACM, 204–213.

[23] Steven S. Skiena. 2008. *The Algorithm Design Manual* (2nd ed.). Springer Publishing Company, Incorporated. https://doi.org/10.1007/978-1-84800-070-4

[24] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L. Greathouse, and Zhiying Wang. 2014. PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), 445–457. https://doi.org/10.1109/MICRO.2014.17

[25] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2012. Load Sharing for Optimistic Parallel Simulations on Multi Core Machines. *SIGMETRICS Perform. Eval. Rev.* 40, 3 (Jan. 2012), 2–11. https://doi.org/10.1145/2425248.2425250

[26] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. *SIGPLAN Not.* 51, 4 (March 2016), 545–559. https://doi.org/10.1145/2954679.2872375