Power Capping: What Works, What Does Not

Pavlos Petoumenos*, Lev Mukhanov†, Zheng Wang‡, Hugh Leather*, Dimitrios S. Nikolopoulos†
*School of Informatics, The University of Edinburgh, UK
{ppetoume, hughleat}@inf.ed.ac.uk
†School of EEECS, Queen's University Belfast, UK
{l.mukhanov, d.nikolopoulos}@qub.ac.uk
‡School of Computing and Communications, Lancaster University, UK
z.wang@lancaster.ac.uk

***Abstract*—Peak power consumption is the first order design constraint of data centers. Though peak power consumption is rarely, if ever, observed, the entire data center facility must prepare for it, leading to inefficient usage of its resources. The most prominent way for addressing this issue is to limit the power consumption of the data center IT facility far below its theoretical peak value. Many approaches have been proposed to achieve that, based on the same small set of enforcement mechanisms, but there has been no corresponding work on systematically examining the advantages and disadvantages of each such mechanism. In the absence of such a study, it is unclear what is the optimal mechanism for a given computing environment, which can lead to unnecessarily poor performance if an inappropriate scheme is used. This paper fills this gap by comparing for the first time five widely used power capping mechanisms under the same hardware/software setting. We also explore possible alternative power capping mechanisms beyond what has been previously proposed and evaluate them under the same setup. We systematically analyze the strengths and weaknesses of each mechanism, in terms of energy efficiency, overhead, and predictable behavior. We show how these mechanisms can be combined in order to implement an optimal power capping mechanism which reduces the slowdown compared to the most widely used mechanism by up to 88%. Our results provide interesting insights regarding the different trade-offs of power capping techniques, which will be useful for designing and implementing highly efficient power capping in the future.**

***Keywords*-Power Optimization; Power Capping; Compiler; DVFS; RAPL**

I. INTRODUCTION

Data centers provide the IT infrastructure that powers many of today's computing environments – from big data and cloud computing to large-scale Internet services. With more and larger data centers being built every year, they are now among the largest consumers of electricity in advanced countries [1], with individual data centers consuming from tens of kilowatts to tens of megawatts [2]. To deal with this high power consumption and even higher power density, data centers are equipped with expensive and extensive power conversion and cooling systems which represent a significant part of the initial investment. According to [3] the power supply system alone costs $10 to $20 for each deployed Watt of peak power, even though this peak power consumption is rarely, or ever, reached [4]. This means that the cooling and power supply systems end up being over-provisioned during typical load conditions, wasting limited capital resources.

The most prominent way to tackle this inefficiency is to set external lower limits to the peak power consumption. With a combination of server-level power limiting mechanisms and system-wide power measurements and management, the power consumption can be guaranteed to stay far below the theoretical peak value [4]. This in turn allows the deployment of many more servers for the same power supply or, conversely, a smaller capital investment for the same number of servers. Using real data from Google's data centers, a recent study [5] shows that such a scheme allows 39% more servers to be deployed with no performance loss.

Most approaches to enforcing power caps at the processor level leverage some form of DVFS and DFS [5]–[7], concurrency throttling [8], or idle states [9], [10], combined with OS-level control. Each one exhibits different advantages: higher efficiency, ease of control, finer granularity, or wider range of enforceable power limits. While many research papers have focused on different ways of using and controlling these mechanisms, little attention has been paid on how these interact with and compare against each other.

Even less papers have focused on hardware-centric power management schemes, exemplified by Intel RAPL [11]. In contrast to software-centric approaches which operate at coarser granularities (typically seconds), these hardware-centric methods can operate at finer granularities (milliseconds to seconds) [12]. Hardware approaches have more access to low-level hardware information and the internals of the DVFS subsystem, offering fine-grained power cap controls and tight feedback-based control loops. As hardware-centric capping mechanisms are becoming ubiquitous on computing systems, it is necessary to understand whether such a scheme is better than other power capping approaches and how we can improve on it when it is not.

This paper attempts to shed some light on the state of power capping. Using a common hardware setting and the same set of representative benchmarks, we evaluate five widely used power limit enforcement mechanisms in terms of achievable performance under the same power budget,

overhead, and predictability. We also explore alternative ways to control the power consumption, examining two mechanisms not previously discussed in the literature as power capping techniques. This is the first comprehensive, quantified analysis for those schemes. Our results provide interesting insights and trade-offs for power capping techniques, which will be useful for future implementations.

The technical contributions of this work are as follows.

- We provide a comprehensive comparison, using representative application workloads, of existing and new power capping mechanisms including DVFS, DFS, RAPL, thread packing, forced idleness, NOP insertion, and compiler-based power control.
- We explain when and why each technique is efficient at limiting the power consumption.
- We examine how different techniques affect each other when used at the same time.
- We identify the most efficient combination of power capping mechanisms.

The remainder of this paper is organized as follows. Section II describes and explains the power management techniques used in this paper, including two not discussed before in the literature. Our experimental setup and results are presented in Sections III and IV respectively. This is followed in Section V by a discussion over the relative advantages of each technique and how they can be combined to enforce power caps in the most efficient way. We close this paper with some concluding remarks.

II. POWER CAPPING TECHNIQUES

This section gives an overview of seven techniques that are evaluated in this work.

DVFS: Dynamic Voltage-Frequency Scaling (DVFS) is the most convenient and commonly used knob for controlling power at the processor level. Using DVFS, performance can be increased by using a higher supply voltage or frequency for hardware components (e.g. processors or memory) and power consumption can be reduced by decreasing the voltage or frequency – often at the cost of longer application runtime. Earlier works used DVFS mainly for minimizing energy consumption, but in the last ten years research has also focused on enforcing power/thermal caps through DVFS [5], [7], [13], [14]. Typical approaches use either online heuristics or simple models to find the optimal frequency without violating the power limit.

DFS: Under dynamic frequency scaling (DFS) runtime and dynamic power consumption scale linearly with frequency, while keeping the dynamic energy consumption roughly constant. While DFS manages to limit the overall power consumption, it does so inefficiently. In modern integration technologies dynamic power consumption represents only a part of the total power consumption, so the power consumption does not scale down as much as the runtime scales up, leading to increased energy consumption.

RAPL: Running Average Power Limit (RAPL) is a management interface provided by Intel processors [11], which combines automatic DVFS and clock throttling in order to keep the power consumption of the processor below a user-defined threshold. RAPL employs an internal model of energy consumption, to estimate the average power consumption over a window of time, and uses DVFS to bring it as close as possible to the power cap and clock throttling to enforce it precisely. Power can be controlled and measured in three distinct domains, including the cores and caches, the entire CPU, or the DRAM.

RAPL improves on plain DVFS in two major ways. First it integrates power monitoring and control inside the chip, making it more accurate and faster to identify and adapt to workload changes. Second, by combining DVFS and clock throttling, RAPL can provide more performance levels than DVFS alone and therefore finer management granularity.

Forced Idleness & Thread Packing: Putting cores into low power idle modes is another way of limiting the power consumption of a program. Multiple techniques depend on it indirectly (e.g. [10], [15], [16]) by reducing the CPU load and enabling the operating system to put some of the cores into an idle mode. Two mechanisms which employ sleep modes more directly as a power capping technique are Forced Idleness [9] and Thread Packing [8]. In the former, the application is periodically forced to pause execution causing the whole processor to go into a low power sleep mode. In the latter, multi-threaded applications are restricted to run on a subset of the available cores, reducing the active concurrency and forcing some of the cores in an idle mode. While offering easily predictable and energy efficient power/performance scaling, both interfere with the communication patterns of the application, in the case of Thread Packing with the communication between threads, in the case of Forced Idleness with the communication between the application and other process, including the OS. Depending on the workload, this can have adverse effects on the performance, which might limit their usefulness.

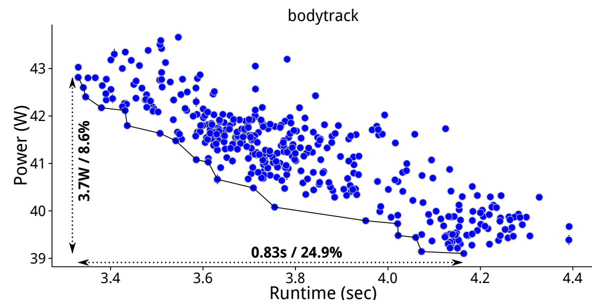

Figure 1. Power consumption vs Execution Time for different compiled versions of `bodytrack` from the PARSEC benchmark suite. The line connects the binaries forming the pareto-optimal front that describes the optimal trade-offs between runtime and power consumption.

Compiler-based power control: Prior work has shown that different compiler optimizations can have significantly different effects on the power consumption of applications, by e.g. changing the IPC of the application, the instruction sequences [17], or loop-level code optimization [18], [19]. In this work, we examine whether this behavior can be exploited in order to enforce power caps. Many of the proposed methods use analytic models or search to generate energy-efficient code through changing the compiler flag (such as the loop unroll factor). In this work, we implemented a typical compiler-based approach similar to MILEPost GCC [20], which first generates many different binaries using different compiler flag settings and then uses profiling runs to identify the binaries forming the pareto-optimal front, i.e. the binaries providing different but optimal trade-offs between performance and power consumption.

As an example, Figure 1 shows 517 unique binaries for `bodytrack`. These binaries are generated using over 2,000 different compiler flag options where identical binaries are removed. The line connecting some of the points is the pareto-optimal front. This front can vary up to 24.9% for the execution time and up to 8.6% for the power consumption for this application. Using this information, a runtime power capping system can determine which binary to use in order to reduce the power consumption below the power limit with as little performance loss as possible.

NOP insertion: Another approach to reducing the power consumption without explicit hardware support is inserting redundant NOP instructions into the binary. Depending on the underlying architecture, NOP instructions either insert bubbles into the pipeline or are replaced with low power instructions. The x86/x86_64 architecture is an example of the former case, with NOP instruction being simply removed at the decode stage, which means that they cause switching activity and power consumption only at the front-end of the core. By inserting NOP instructions at compile time, we can thus limit the power consumption. NOP insertion can be seen as a benchmark of the efficiency of the compiler-based approach: if it achieves similar trade-offs between power and performance, then the compiler reduces power just by executing more instructions, if not, the compiler changes the binary and its power consumption in more complex and potentially interesting ways.

III. Experimental Setup

This section describes the details of the experimental case studies that we undertook, including the platform and benchmarks used, the implementation details of the power capping techniques, and the evaluation methodology.

A. Hardware Platform

We evaluated our approach on a server platform with one Intel Xeon E5-2650 processor and 64 GB DDR3 RAM. Hyper-threading and Turbo Boost were disabled. DVFS is controlled through the Linux `cpufreq` module, while RAPL limits are set using the `power_gov` program provided by Intel. On the software side, the server runs CentOS 6.5 (kernel-2.6.32). We used gcc 4.8 with the best-found compiler options for compiling C/C++ applications. A brief overview of the platform's characteristics is given in Table I.

Table I
HARDWARE AND SOFTWARE CONFIGURATIONS

Processor	Intel Xeon E5-2650, 8 cores, 32KB/32KB I/D-Cache per core 2MB L2 cache, 20MB L3 cache
DVFS	2.0GHz Max Frequency (no turbo), 1.2GHz Min Frequency
Memory	64 GB DDR3
O.S.	CentOS 6.5 with kernel 2.6.32
Compiler	gcc 4.8.2 -march=native

Table II
BENCHMARKS, INPUTS AND RUNTIME/POWER CONSUMPTION UNDER NO POWER CAP

Benchmark	Suite	Input	Time (sec)	Power (W)
kmeans	MineBench	"edge"	7 - 15	31.4 - 34.3
x264	PARSEC	"native"	20 - 44	29.5 - 33.6
bodytrack	PARSEC	"native"	30 - 36	25.0 - 27.5
blender	Blender	cornellbox	72 - 79	31.7 - 32.5

B. Benchmarks

We selected the multi-threaded applications among those suggested by [21] as representatives of typical data center workloads. These applications include kmeans from the MineBench suite, the application Blender, and x264 and bodytrack from PARSEC. We run all benchmarks with eight threads, one thread per physical core. We used inputs provided by the benchmark suites that are most similar to the ones suggested by [21]. Since with these inputs of the benchmarks can take up to one minute to run, performing many profiling runs will take unacceptably long time. This makes it impractical for the compiler-based technique to construct the pareto-optimal front of power and runtime (as described in Section II) using these inputs. To mitigate the problem, we used smaller inputs to search for the most power-efficient binary for a given performance target instead. More details about the benchmarks and their inputs can be found in Table II.

C. Experimental methodology

The aim of this work is to compare the efficiency of different power control mechanisms: DVFS, DFS, RAPL, forced idleness, thread packing, compiler-based power control, and NOP insertion. To achieve this, we executed all our benchmarks under multiple settings for the control knobs (detailed in Section III-D) of these mechanisms and compared the measured runtime and power consumption.

For the comparison between the different techniques to be clear and fair, we need their highest performance/power state to be the same. Since for the compiler-based technique that state is using the fastest binary found during the profiling

Table III
CONTROL MECHANISMS PARAMETERS

	DVFS/DFS	RAPL	Core count	Idleness (msec)	NOPs
kmeans	1.2 - 2.0GHz	12-36W	1-8	5-38/50	1 - 512
x264	1.2 - 2.0GHz	12-36W	1-8	5-38/50	1 - 13
bodytrack	1.2 - 2.0GHz	12-32W	1-8	5-38/50	50 - 1000
blender	1.2 - 2.0GHz	12-34W	1-8	5-38/50	1 - 64

runs, we used the fastest binary for all techniques. This ensures all schemes use the same, highly optimized binary.

We performed energy measurements using ALEA [22], an in-house energy profiler based on statistical sampling of power consumption. The profiler operates in user space and provides fine-grain energy profiling at the granularity of basic blocks. On our Xeon platform, the profiler interfaces with the RAPL MSRs (Machine Specific Registers) to perform energy accounting, which gives us fine-grained, highly-accurate energy information. Runtime measurements were done through the C standard library function `gettimeofday`. All measurements on real hardware have some inherent amount of noise that we have to deal with. The problem becomes even worse with parallel applications which often have variable runtime due to the non-determinism of thread ordering and communication. For this reason, all our experiments were repeated enough times until the 95% confidence interval around runtime and power falls under 0.5% of the expected value, with an upper limit of 80 repeated executions.

D. Control knobs parameters

A brief overview of the parameters of the power control mechanisms used in our experiments is provided in Table III. More specifically, the control mechanism parameters are the following:

DVFS & DFS: Frequency for DVFS ranges from the maximum frequency without turbo (2.0GHz) to the minimum (1.2GHz) in steps of 100MHz.

RAPL: The power limit for RAPL varies from the maximum power consumption observed during the DVFS experiments down to 12 Watts in steps of 1 Watt. While RAPL in Sandy Bridge servers monitors three power planes (Package, `PP0` and `DRAM`), in this work we only control `PP0` which includes the core and the caches.

Forced Idleness & Thread Packing: Idleness was enforced using a helper program which periodically sent `SIGSTOP` and `SIGCONT` signals to the benchmark process. The period of the helper's program main loop was 50 milliseconds and the time spent with the application stopped during each cycle varied from 5 to 38 milliseconds. For thread packing, we varied the number of active cores from one to eight, in steps of one core.

Compiler-based Power Control: We used the binaries which form the pareto-optimal front in the experiments as described in Section II. For the search of the pareto-optimal sets of compiler optimizations, we performed two searches using a genetic algorithm, one optimizes for execution time, another optimizes for power.

NOP Insertion: We added various numbers of `NOP` instructions in the hot loops of the benchmarks. The number of NOP instructions was chosen empirically so that the range of measured runtimes would be close to the range observed in the DVFS based experiments.

IV. RESULTS

In this section we present the results of our evaluation. First we compare the efficiency of each mechanism by investigating the performance slowdown under the same power budget. Next, we quantify the overhead introduced by each mechanism. Finally, we examine how predictable they are in affecting power and performance.

A. Comparison of efficiency

Figure 2 compares the runtime for seven power capping mechanisms (as described in Section II) under different static power caps, while Figure 3 focuses on the upper left corner to better illustrate the differences between the mechanisms for high power limits. All parameters affecting the power consumption (e.g. frequency, cores), except for the one used by each mechanism, are fixed to their default value.

DVFS & DFS: As a commonly used power management scheme, DVFS is the overall winner in terms of efficiency. No other mechanism gives faster application performance within the same power budget when compared to DVFS. As it can be deduced by comparing DVFS with DFS, it is voltage scaling which makes DVFS efficient. Changing the voltage reduces not only the amount of activity per unit of time like other mechanisms, but also the cost of each instruction, something that no other mechanism can achieve. However, DVFS still suffers from two drawbacks: coarse granularity and limited range. Specifically, on our experimental setup, DVFS offers only nine distinct operating points and it cannot reduce the power consumption to less than 17 Watts. If we need to restrict the power consumption more than that or we need finer power control, we need to augment DVFS with other power capping mechanisms.

RAPL: By combining DVFS and clock throttling RAPL can get the best of both techniques. While the power limit lies within the range supported by DVFS, RAPL relies mainly on it alone and achieves similar performance to DVFS. Outside that range, RAPL is still able to enforce power caps, even if its efficiency in doing so is much lower. In some cases, RAPL starts behaving significantly worse than DVFS for power caps towards the lower end of the DVFS-supported range. The reason for that is power spikes: while on average the power consumption stays below the requested power limit using the lowest voltage-frequency level, the actual running average fluctuates enough for the power limit to be violated for long periods of time. As a response, RAPL introduces, at a cost, clock throttling to limit the

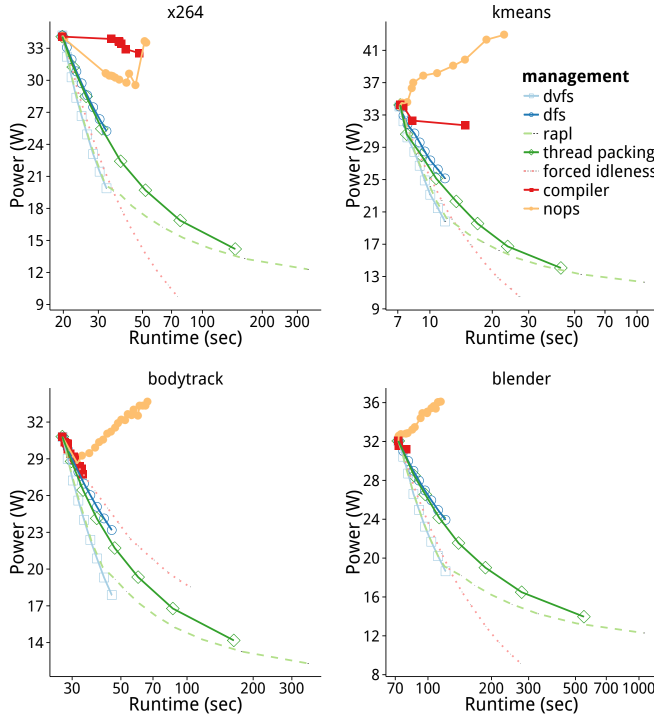

Figure 2. Power consumption vs Execution Time with various power control mechanisms for four different benchmarks. Each curve represents one mechanism. All curves start from the same point on the left side which represents the case of no power limit. The closer to the start of the axes a curve, the more energy efficient the mechanism is.

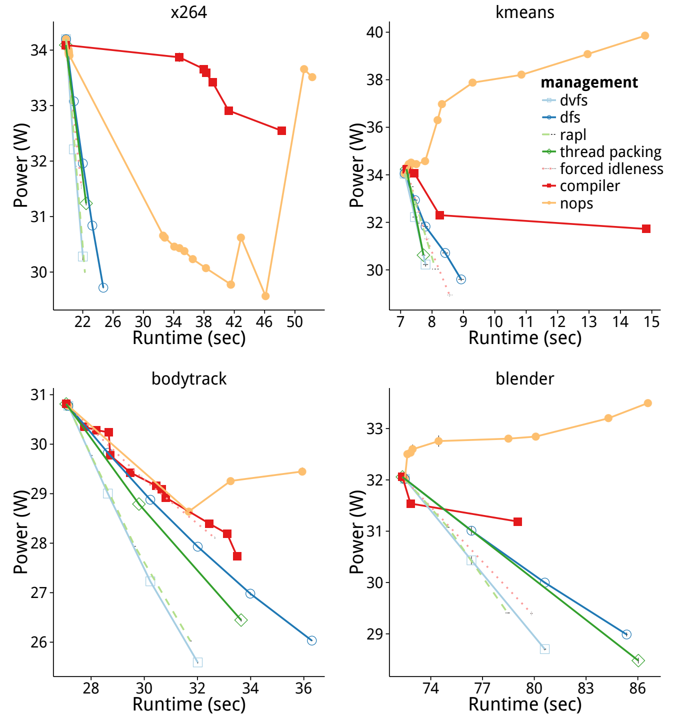

Figure 3. Detail of the Figure 2 subplots.

power consumption during the peaks. At the worst case, for `bodytrack`, runtime under management with RAPL can be up to 16.7% higher than under DVFS, while at the best case, for `x264`, the additional slowdown never exceeds 10%.

Forced Idleness: As seen in Figure 2 for all benchmarks except `bodytrack` forced idleness performs just slightly worse than DVFS, always within 10% of the runtime under DVFS. At the same time it produces a much wider range of enforceable power limits, from 30+ down to a couple of Watts, and does so with a very fine granularity. The reason for its efficiency is its high power proportionality. While most mechanisms offer performance proportional to the *dynamic* power consumption, forced idleness scales proportionally both the dynamic and the static power consumption by switching between the most energy efficient on state and the lowest power off state, which in our case consumes about 1 Watt. The only benchmark were forced idleness fails to achieve that is `bodytrack`. The reason for that is that IO requests initiated by the application keep the processor busy even after the application is stopped. So while the application process itself is idle, the processor does not stay idle long enough to enter a power saving sleep mode. A more sophisticated forced idleness implementation should be able to deal with this issue, but we wanted to keep our implementation as simple and close to the original as possible.

Thread Packing: This technique has a similar but slightly better effect on power over DFS. By leaving whole cores idle, both the dynamic power consumption and the performance are almost proportionally affected, similarly to DFS. For `x264`, deactivating each core reduces the power consumption by 2.7 to 3.0 Watts, while performance is reduced by 12% to 12.6% compared to the case of 8 cores. How much thread packing can outperform DFS depends on the scalability of the workload. For example, `kmeans` does not scale very well, especially when going from 7 to 8 cores, so thread packing can limit the power consumption without a proportional effect on performance.

Compiler-based Power Control: As we see in Figure 3, the compiler does provide us with some control over the power consumption, with the range of enforceable power limits being between 4 Watts for `x264` to less than 1 Watt for `blender`, but it is not a very efficient way to control power. For `x264` and `kmeans` the compiler produces a worse trade-off between performance and power than every other mechanism and in the case of `x264` it is even worse than just inserting NOP instructions in the benchmark binary. For `blender` and `bodytrack` the results are slightly better. For the former, changing the power consumption from 32.1 to 31.3 Watts can be achieved with less performance degradation than DVFS, while for the latter the performance achieved is similar to that of forced idleness. Overall, this compiler-based technique can be occasionally useful to control power, but its narrow range of enforceable power limits and low efficiency mean that it cannot be used on its

own.

NOP-insertion: Inserting NOP instructions is, as expected, a very inefficient way to control power. For two of our benchmarks, `blender` and `kmeans`, it fails to reduce the power consumption at all, while for the other two it does to a limited degree but with much higher performance degradation than the other mechanisms examined. Since NOP instructions are executed without affecting the processor's state or creating dependencies, they just consume extra energy with little impact on performance and "useful" power consumption.

B. Comparison of overhead

It is also important to understand the runtime overhead of each mechanism as a good scheme should have little overhead. To do so, we executed `x264` while switching between the highest and the lowest performance state for each mechanism every 100 milliseconds. Additionally, we executed the benchmark using the same setup but with no switching, every 100 milliseconds resetting the same performance state, either the highest or the lowest, in order to measure the runtime for those states. By comparing the runtimes between the case with switching and the cases with no switching, we can estimate the runtime overhead introduced by each of the power control mechanisms. For all cases, we repeated each experiment enough times for the confidence interval of its runtime to fall below 0.01%.

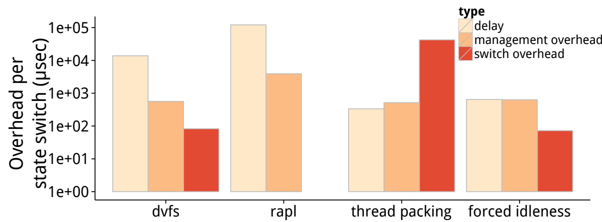

Figure 4. Delay and Overhead per power/performance state switch for DVFS, forced idleness, RAPL, thread packing when running `x264`.

Figure 4 shows the runtime overhead for each mechanism when running `x264`. The first bar of each group represents the delay between initiating a state switch and completing it, the second represents the runtime overhead introduced by the power management actions even when they do not affect the performance state, and the third bar represents the runtime overhead solely due to state switches.

For DVFS and forced idleness both the switching and the management overhead are similarly low, 70-90 microseconds for the switching overhead and 550-650 microseconds for the management overhead. Thread packing has the lowest delay in changing the state and introduces little management overhead, but the actual process of continuously preempting and moving the application's threads has a high cost, on average 42 milliseconds per switch. For RAPL we did not estimate its switching overhead. Its relatively long average delay (about 120 milliseconds) introduces significant amounts of noise into the lengths of the high performance and low performance parts of the 200 milliseconds cycle, making it impossible to estimate accurately the switching overhead. Still, since RAPL is based on DVFS and DFS, we would expect it to have a similarly low switching overhead. While its non-switching management overhead is high, almost 4 milliseconds per switch, the actual overhead in a more realistic scenario should be low: RAPL does not need software intervention to adapt to changing workloads or power spikes, which means that software initiated management actions are very infrequent, only when the power cap has to change.

C. Predictability of power and performance scaling

For each power capping mechanism, we would like to determine how performance and power are affected by the actual value of control knob of the technique and whether the effect can be easily predicted or not. If the effect is predictable, that means that we can accurately scale the power consumption below any given power limit, while knowing exactly how performance will change. In turn this allows us to choose which mechanism to use to enforce a power limit, just by calculating their predicted effect on performance and choosing the one which hurts it the least, without having to first measure the effect.

Figure 5 depicts how power consumption and performance are affected by the control knob of each technique. For RAPL that knob is the requested power consumption, for DVFS and DFS it is the frequency, for forced idleness it is the percentage of time spent in the active state, for thread packing is the number of active cores, and for the compiler-based technique it is the power/performance measured during the search phase using a different input. Both the x and the y-axis are normalized to the maximum value of that axis.

1) Power Modeling: Regarding power, for all techniques except the compiler-based one there is a strong linear relation between the value of the main power knob of the technique and the enforced power consumption, with the coefficient of determination R^2 ranging from 99.74% to 100%. Exactly how the scaling of each technique's main knob and the scaling of the power consumption are connected is presented in Table IV. These linear relationships fit well with how each mechanism affects power. Dynamic power consumption, which represents 64-65% of the total power consumption, scales linearly with frequency scaling and core deactivation, while static power consumption (35-36% of the total) remains unaffected. DVFS causes the dynamic power consumption to scale quadratically, but for the limited range of voltage and frequency levels supported by our processor the relationship can be modeled as a linear scaling of the whole power consumption, static and dynamic. Forced idleness is modeled as a straight line connecting the full-on power consumption and the power consumption while in the idle state, 46% of the total for `bodytrack`, 4% of the

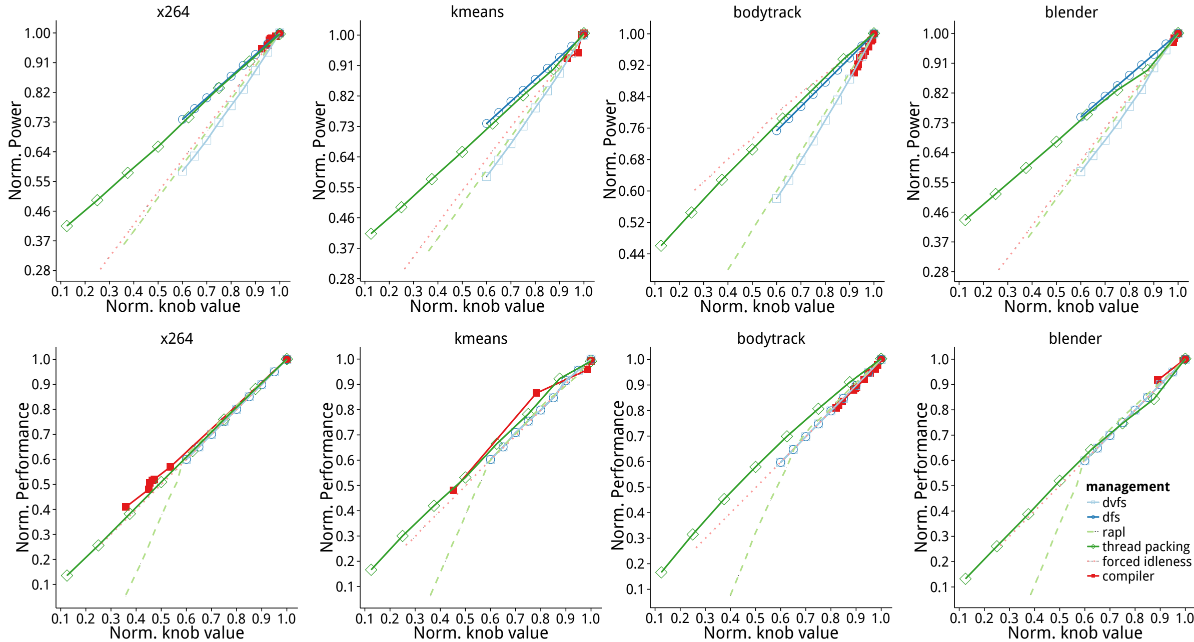

Figure 5. Power knob value vs power consumption (top) and performance (bottom) for all the power control techniques. Both axes are normalized to their maximum values. Straight lines indicate strong linear correlation between knob value scaling and power/performance scaling.

Table IV
RELATIONSHIP BETWEEN SCALING OF THE CONTROL KNOB VALUE AND SCALING OF THE POWER CONSUMPTION/PERFORMANCE FOR ALL MECHANISMS

	power	performance
DFS	0.64 * knob + 0.36	1.0017 * knob - 0.001
DVFS	1.05 * knob - 0.05	1.0016 * knob - 0.001
RAPL	1.00 * knob - 0.00	1.4707 * knob - 0.385
forced idleness (all)	0.86 * knob + 0.14	0.9991 * knob
forced idleness (bodytrack)	0.54 * knob + 0.46	
forced idleness (rest)	0.96 * knob + 0.04	
thread packing	0.65 * knob + 0.35	0.9707 * knob + 0.042
compiler-based management	low correlation	0.9046 * knob + 0.087
NOP-insertion	low correlation	low correlation

total for the other benchmarks. And RAPL enforces almost perfectly the requested power limit, as it was expected.

The compiler-based power management technique does not behave as orderly. As we see in Figure 5, there is high correlation between the previously seen power consumption scaling and the one seen during the evaluation only for `bodytrack` and `blender`. For `x264` and `kmeans` the power consumption scaling is more unpredictable. This means that using the compiler to control the power consumption is not straightforward: we have to first determine how changing the application binary affects the power consumption, by testing all the binary versions.

In Figure 6 we see in more detail how accurately we can predict their effect on power. The average error is less than 0.5% (0.16 Watts) and the worst case error less than 1.0% (0.31 Watts) for DFS, DVFS, and RAPL. Thread packing and forced idleness have slightly less predictability. The model for thread packing produces 1.5% average and 3.6% worst case error, while for forced idleness the error can reach 6.8% on average and 23% in the worst case. The reason for this behavior is the failure of forced idleness in putting the processor into a sleep mode, when `bodytrack` is executed. As seen in Table IV, for the other three benchmarks the idle state is a low power sleep state with the power consumption around 1.2 Watts, but for `bodytrack` this "idle" state consumes almost 14 Watts, making it impossible to describe the effects of forced idleness with a single linear model. Treating separately `bodytrack` from the other benchmarks allows us to predict the scaling of the power consumption accurately, as seen in the fourth and fifth bar of Figure 6. Overall, these results mean that it is easy to predict with high accuracy how DFS, DVFS, RAPL, and thread packing will affect the power consumption, which allows these technique to safely enforce power caps. Forced idleness can be more difficult to use safely.

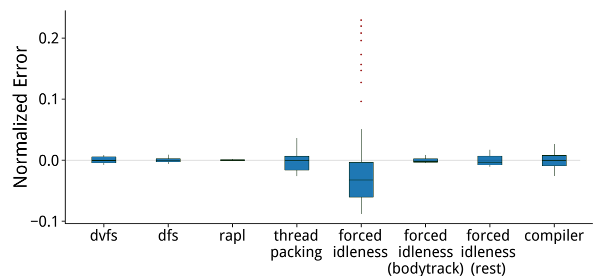

Figure 6. Error when modeling the relationship between the power knob value and the power consumption as linear. The error is normalized to the maximum power consumption for each benchmark. Boxes cover the area between the first and third quartiles of the error distribution (Interquartile Range/IQR). The horizontal line inside the box indicates the median error. Red dots show the outliers.

2) Performance Modeling: Again most of the mechanisms display a strong linear relation between knob value and performance scaling, with all of them except RAPL exhibiting a coefficient of determination between 99.2% to 100%. This is especially true for DFS, DVFS, and forced idleness which scale performance almost the same as they scale their knob values (Table IV). Thread packing displays some non-linearity, mainly for `bodytrack` and `kmeans`. Depending on the communication patterns of the threads, restricting them on a fewer number of cores might reduce the communication overheads and therefore cause the impact on performance to be less than expected. While beneficial, this also means that it is more difficult to predict accurately how power capping will affect performance. Compiler-based management produces better results when predicting performance compared to predicting power consumption. Finally, RAPL displays non-linear behavior. While it uses two mechanisms, DVFS and DFS, with easily predictable effects on performance, using a higher level control knob, the power limit, obfuscates this predictability.

Summary: Overall, DFS and DVFS are the two most predictable power control mechanisms in terms of effects both on performance and power consumption. Forced idleness is similarly predictable as long as we can determine whether it will manage to force the processor into a sleep mode or not. Thread packing exhibits more variability. RAPL scales the power consumption perfectly, as expected, but its effects on performance are harder to determine beforehand. Finally, the compiler-based technique can be unpredictable in terms of power scaling but its effects on performance are more easily predicted.

V. IDEAL POWER CAPPING

As we saw in the previous section each power capping mechanism has its own advantages and disadvantages. DVFS is clearly the most efficient way to control power but has limited range of enforceable power limits. As integration technology scales down, the gap between the supply voltage and the threshold voltage will continue to narrow, limiting further the range of power limits enforceable by DVFS. DFS is fast, fine-grained, and provides a wide range of enforceable power levels, but is inefficient. RAPL combines the benefits, but also the drawbacks, of the two techniques: fast, fine-grained control, but efficient only in the same range of power limits as DVFS. Forced idleness can be anything between as efficient as DVFS and as inefficient as DFS depending on the application, with a wide range of enforceable power limits. Thread packing behaves better than DFS and has consistent and predictable effects on power, but it incurs significant overheads and is not as fine-grained as the other techniques or as efficient as DVFS, RAPL, and (usually) forced idleness. Compiler-assisted management can occasionally outperform DVFS and provide a wide range of power levels, but usually does not.

A better power capping technique should be able to combine the benefits of all these mechanisms: fast, fine-grained, with as little performance degradation as possible, and predictable. RAPL already does a very good job, failing only in terms of performance for low power limits, so it is reasonable to assume that any near optimal technique could build on top of RAPL (or any similar mechanism). To deal with low power limits, we can combine it with any mechanism orthogonal to DVFS, providing low performance degradation, and able to enforce low power limits. The most appropriate mechanisms for this task are forced idleness and thread packing. In the rest of this section, we combine RAPL with other power capping mechanism and compare the results of the combined techniques.

RAPL + Forced Idleness/Thread packing: In Figures 7 and 8 we see how combining RAPL and either forced idleness or thread packing affects performance and power, with the different curves representing management under RAPL for different cores counts or time spent paused. For thread packing, the curves cross each other, their relative efficiency in controlling power changing as the power limit changes. For the higher power limits it is almost always better to use eight threads, meaning that thread packing is disabled, except for `kmeans` where using seven instead of eight threads increases the energy efficiency of the benchmark. For lower power limits, the optimal number of cores to use changes steadily towards lower values, until eventually for power limits below 10-9 Watts packing all the thread in one core becomes the optimal management decision. Similarly for forced idleness the optimal ratio of time spent in the on state changes with the power limit, from always on for the highest limits to almost always off for the lowest limits. Again the exception is `bodytrack` where forced idleness fails to put the processor into a sleep mode, which makes disabling forced idleness always preferable. For both combinations the optimal policy is roughly as follows: while RAPL power limits are enforced primarily through DVFS, keep forced idleness and thread packing disabled, and beyond that use the additional mechanisms to keep RAPL from using clock throttling.

RAPL + Force Idleness/Thread Packing/Compiler-based Technique: In Figure 9 we see how combining RAPL with forced idleness, thread packing, and compiler-based management compares against RAPL for our evaluation benchmarks. The x-axis is the power consumption, while the y-axis is the runtime achieved for this power consumption, normalized to the runtime under RAPL. The curves represent the pareto-optimal front of power-runtime points presented in the previous figures. Almost all mechanisms behave identically for high power limits, since they use exclusively DVFS in this range to control power. The exception is `kmeans` where, as we saw before, using seven instead of eight cores is more efficient, so RAPL combined with thread packing outperforms the other mechanisms. For power limits

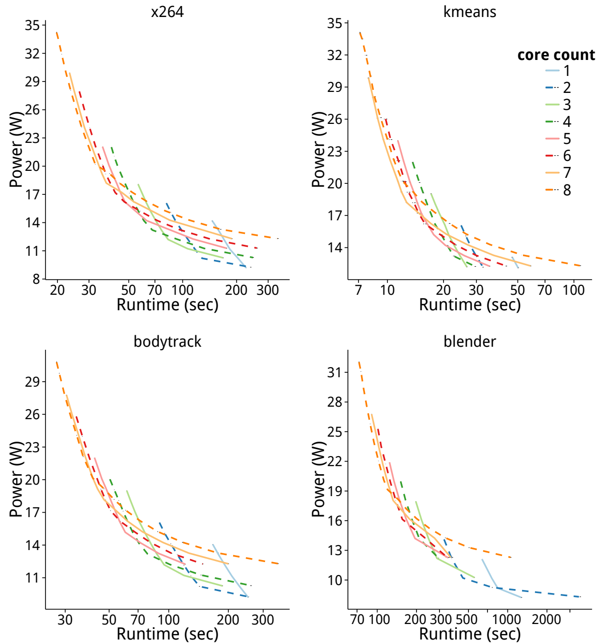

Figure 7. Power consumption vs Execution Time for RAPL with thread packing. Each curve corresponds to a different number of active cores. The optimal number of cores is decreased as the power limit is reduced.

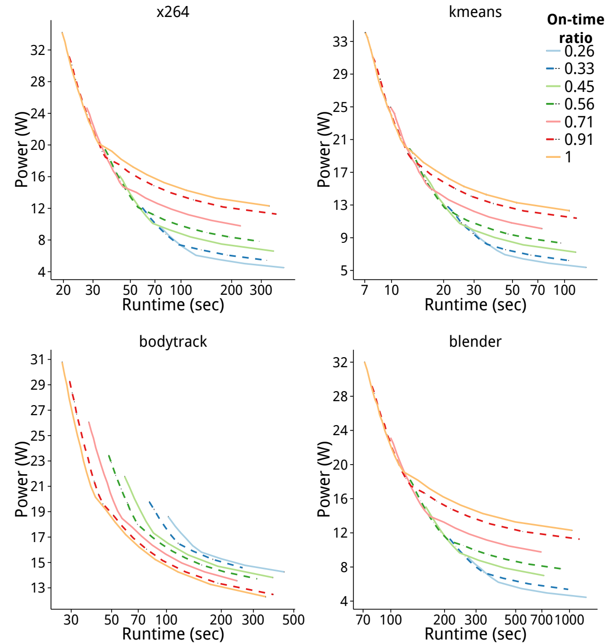

Figure 8. Power consumption vs Execution Time for RAPL combined with forced idleness for different periods of the application being paused. For all benchmarks, except `bodytrack`, the optimal amount of time spent paused is increased as the power limit decreases.

below those enforced by DVFS, forced idleness usually outperforms by far all other mechanisms, both in terms of achievable performance under the same power cap (up to 5x faster than RAPL, up to 72% faster than RAPL + thread packing) and in terms of lowest enforceable limit (down to 4-5 Watts). Only for `bodytrack` do the results differ with thread packing being more efficient. RAPL combined with compiler-assisted management behaves almost always identically to RAPL: with compiler-assisted management being less efficient than DFS, we prefer to reduce the power consumption with clock throttling than by using the compiler. The sole exception is `blender`. There we manage to enforce the power limits between 10 and 17 Watts with less performance degradation than RAPL, in the best case with up to 21% less runtime.

Best Combined Technique: Overall, RAPL combined with forced idleness seems to be the best out of the three combined policies: both of them are fast, low-overhead mechanisms and they produce the best results for three out of our four benchmarks. Thread packing is still useful for the cases where forced idleness does not work as intended or for the cases where using the maximum number of cores is not the most energy efficient choice. Considering their relative advantages and disadvantages, the ideal policy should use all three mechanisms in complementary roles of different granularity. Thread packing for infrequent decisions based on long term power limit trends and maximizing the energy efficiency, forced idleness for management at the millisecond scale aiming at keeping RAPL from using clock throttling, and finally RAPL enforcing the power limit precisely by constant monitoring of the power consumption.

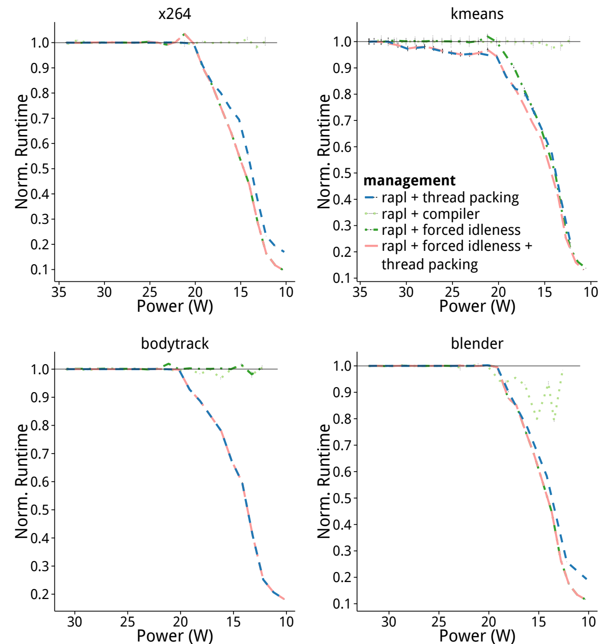

Figure 9. Power consumption vs Runtime normalized to the one of RAPL, for RAPL combined with thread packing, RAPL combined with compiler-based management,RAPL combined with forced idleness and RAPL combined with thread packing *and* forced idleness.

How this policy would fare is presented by the last curve of Figure 9. As expected the ideal power capping policy

matches the performance of the best mechanism for `x264`, `bodytrack`, and `blender`. When `kmeans` is executed, all three underlying policies help enforce the power limit, allowing the ideal policy to outperform all the others. Overall, it manages to limit the power consumption with up to less 88% slowdown than RAPL, 72% than RAPL with thread packing, and 83% than RAPL with forced idleness.

VI. Conclusions

In this paper we systematically examined five prominent power capping mechanisms and two mechanisms not previously discussed in the literature, compiler-based power capping and NOP insertion. Our study investigates not only the relative efficiency of these mechanisms in enforcing power limits, but also the predictability of their effects on performance and power as well as the overheads associated with each technique. Our results provide valuable insights, showing that RAPL and DVFS are highly effective and fast power capping techniques when the desired power limit is high enough, while forced idleness and thread packing are usually a better choice for low power limits. Clock throttling and DFS while widely used, e.g. in RAPL, are among the worst performing techniques. Alternative mechanisms, like the compiler-based management or NOP insertion, are unsuitable as power capping techniques both in terms of effectiveness and predictability. Overall, the results presented in this paper suggest that to fully optimize power capping, we need to use RAPL together with thread packing and forced idleness, in a coordinated way which fully exploits their advantages, while minimizing the effects of their disadvantages.

Acknowledgments

This work has been supported by the UK Engineering and Physical Sciences Research Council under grants EP/L000055/1, EP/H044752/1 (ALEA), EP/L004232/1 (ENPOWER), EP/M01567X/1 (SANDeRs), EP/K017594/1 (GEMSCLAIM), EP/M015823/1, EP/M015793/1 (DIVIDEND), EP/M015742/1, and by the European Commission under Grant Agreements FP7-610509 (NanoStreams) and H2020-644312 (RAPID).

References

[1] J. G. Koomey. (2011, Aug.) Growth in data center electricity use 2005 to 2010. [Online]. Available: http://www.analyticspress.com/datacenters.html

[2] R. Brown *et al.*, "Report to congress on server and data center energy efficiency: Public law 109-431," U.S. Enviromental Protection Agency, Tech. Rep., Aug. 2007.

[3] W. P. Turner *et al.*, "Tier classifications define site infrastructure performance," The Uptime Institute, Tech. Rep., 2006.

[4] L. A. Barroso, J. Clidaras, and U. Hlzle, *The datacenter as a computer: An introduction to the design of warehouse-scale machines*, 2nd ed., ser. Synthesis lectures on computer architecture. Morgan & Claypool, 2013.

[5] X. Fan, W. D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 13–23.

[6] M. E. Femal and V. W. Freeh, "Boosting data center performance through non-uniform power allocation," in *ICAC '15*. IEEE, 2005, pp. 250–261.

[7] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy, "Optimal power allocation in server farms," in *ACM SIGMETRICS Performance Evaluation Review*. ACM, 2009.

[8] R. Cochran *et al.*, "Pack & cap: adaptive dvfs and thread packing under power caps," in *Micro '11*. ACM, 2011, pp. 175–185.

[9] A. Gandhi *et al.*, "Power capping via forced idleness," 2009.

[10] A. A. Bhattacharya, D. Culler, A. Kansal, S. Govindan, and S. Sankar, "The need for speed and stability in data center power capping," *Sustainable Computing: Informatics and Systems*, vol. 3, no. 3, pp. 183–193, 2013.

[11] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, December 2009, no. 253669-033US.

[12] R. Raghavendra *et al.*, "No "power" struggles: Coordinated multi-level power management for the data center," in *ASPLOS XIII*, 2008.

[13] M. E. Femal and V. W. Freeh, "Safe overprovisioning: Using power limits to increase aggregate throughput," in *Power-Aware Computer Systems*. Springer, 2005, pp. 150–164.

[14] K. Rajamani *et al.*, "Application-aware power management," in *IISWC '06*. IEEE, 2006, pp. 39–48.

[15] R. Nathuji and K. Schwan, "Virtualpower: coordinated power management in virtualized enterprise systems," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 265–278.

[16] A. Kansal *et al.*, "Virtual machine power metering and provisioning," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 39–50.

[17] P. Balaprakash, A. Tiwari, and S. M. Wild, "Multi objective optimization of hpc kernels for performance, power, and energy," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*. Springer, 2014, pp. 239–260.

[18] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of compiler optimizations on system power," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 9, no. 6, Dec. 2001.

[19] J. Seng and D. Tullsen, "The effect of compiler optimizations on pentium 4 power consumption," in *INTERACT-7*, 2003.

[20] G. Fursin *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *International Journal of Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011.

[21] S. Polfliet, F. Ryckbosch, and L. Eeckhout, "Optimizing the datacenter for data-centric workloads," in *Supercomputing '11*. ACM, 2011.

[22] L. Mukhanov, D. S. Nikolopoulos, and B. R. de Supinski, "ALEA: fine-grain energy profiling with basic block sampling," *CoRR*, vol. abs/1504.00825, 2015. [Online]. Available: http://arxiv.org/abs/1504.00825