Per-Application Power Delivery

Akhil Guliani
University of Wisconsin-Madison, USA
guliani@cs.wisc.edu

Michael M. Swift
University of Wisconsin-Madison, USA
swift@cs.wisc.edu

Abstract

Datacenter servers are often under-provisioned for peak power consumption due to the substantial cost of providing power. When there is insufficient power for the workload, servers can lower voltage and frequency levels to reduce consumption, but at the cost of performance. Current processors provide power limiting mechanisms, but they generally apply uniformly to all CPUs on a chip. For servers running heterogeneous jobs, though, it is necessary to differentiate the power provided to different jobs. This prevents interference when a job may be throttled by another job hitting a power limit. While some recent CPUs support per-CPU power management, there are no clear policies on how to distribute power between applications. Current hardware power limiters, such as Intel's RAPL throttle the fastest core first, which harms high-priority applications.

In this work, we propose and evaluate priority-based and share-based policies to deliver differential power to applications executing on a single socket in a server. For share-based policies, we design and evaluate policies using shares of power, shares of frequency, and shares of performance. These variations have different hardware and software requirements, and different results. Our results show that power shares have the worst performance isolation, and that frequency shares are both simpler and generally perform better than performance shares.

CCS Concepts • **Hardware** → **Platform power issues**; *Enterprise level and data centers power issues*;

Keywords Power Management, DVFS, Proportional Shares

ACM Reference Format:
Akhil Guliani and Michael M. Swift. 2019. Per-Application Power Delivery. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303981

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303981

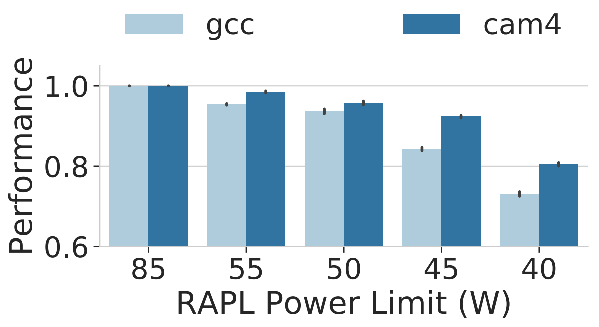


Figure 1. Performance interference between applications with RAPL, normalized to standalone execution at 85W. Application *gcc* is low demand and *cam4* is high demand.

1 Introduction

Power has become a first-class citizen in modern data centers: it is a primary cost in provisioning data centers [16], which emphasize improving overall efficiency. It is a major concern for server designers [21], who have focused on efficiency and power proportionality [5]. It is also a major concern for data-center operators, who must ensure their systems stay within provisioned power limits [18, 28, 56].

Across systems, it is currently possible to provision more power to some hosts than others, such as using more power on hosts executing more applications. The distribution of power is accomplished explicitly through power-aware job placement [56] and implicitly by normal job scheduling.

Within a single host, modern processors provide a rich toolset for managing available power. This toolset is only getting richer as power management rises in importance. For example, some Intel and AMD processors provide per-core dynamic voltage and frequency scaling (DVFS) in hardware, automatic enforcement and reporting of power limits (running average power limit or RAPL [22]), and hardware performance hints (hardware-managed P-states or HWP [1]).

Currently, much of the software leveraging these hardware capabilities target mobile systems where managing battery lifetime (energy) and thermals (temperature) is of prime importance. For example, Linux's *cpufreq* and power governors are largely used by mobile systems. However, in cloud and internet-service data centers there is a large class of heterogeneous workload machines that operate under a power budget due to the high cost of power provisioning [16]. Tools that use these mechanisms to control application power consumption, can benefit such systems.

As a motivating example, Figure 1 shows the impact of Intel processors' current power limiting mechanism, RAPL. This figure shows the relative performance of two different applications running concurrently on different cores of a Skylake processor when subjected to a power limit with RAPL. Further, *gcc* is *low demand* (i.e., uses less power at a given frequency), while *cam4* is *high demand* (i.e., uses more). Also, *cam4* uses Intel AVX instructions, which limits it to a lower maximum frequency of 1667 MHz [29], as compared to 2360 MHz for *gcc*.

When these two applications run together under progressively lower power limits, the processor start throttling the frequency of *gcc*, even though it consumes less power than *cam4*. At 50W, *gcc* is throttled to 1975 MHz (12% reduction) while *cam4* is throttled only to 1570 MHz (5%). At 40W both are throttled to the same frequency of 1240 MHz, but this represents a 48% reduction in frequency for *gcc* but only a 25% reduction for *cam4*. These results show that current power-limiting mechanisms do not address the difference in power usage and frequency limits across cores. Furthermore, any administrator-provided priorities, such as a preference to run *gcc* faster while the power-hungry *cam4* sacrifices performance at lower power levels, are ignored. This makes co-locating latency-sensitive and batch tasks difficult, as batch workloads may exceed power caps and affect the other workloads [33].

To address this problem, we focus on the problem of *differential power delivery*: how can and should a system allocate different amounts of power to different applications to effect prioritization and isolation? We study the mechanisms available in modern processors and develop two classes of differential power-delivery policies. First, we investigate a simple two-level priority model with foreground and background tasks, with the goal that foreground applications get peak performance subject to a power limit. Background applications receive the rest.

Second, we look at a proportional share model, where power is distributed according to shares [55]. Within this class of policies, we investigate allocating shares of three different resources: power, performance, and frequency. These three share types have different hardware and software demands, and result in different, yet useful behavior. For example, distributing power directly requires per-core power measurements, which are not available on Intel platforms. Distributing performance requires a reasonable estimate of application performance in hardware.

In our evaluation on recent AMD and Intel platforms, we found that all policies were effective at providing some level of power isolation across applications. While power shares are conceptually simpler, they are poor at isolating performance. Frequency shares performed similarly to performance shares yet are simpler to implement and require less overhead.

2 Background

In this section we describe the various power management and capping mechanisms implemented by recent microprocessors. We focus on power (energy per unit time) which is of greater importance to datacenters as opposed to energy efficiency, which is often the focus for mobile systems.

2.1 Power Management

The following mechanisms are provided by microprocessors to control power consumption.

Dynamic voltage and frequency scaling (DVFS): DVFS varies the frequency of a processor. With reduced frequency, voltage can be dropped leading to a cubic drop in power and higher energy efficiency. This occurs because dynamic power (P_{dyn}) is related to voltage (V) and frequency (f) according to the equation $P_{dyn} \propto V^2 f$. Current implementations of DVFS are fast (taking 1-30μsec [35]). DVFS has been adopted by almost all current processors [2, 10, 20] and is the preferred mechanism for power management. We see two implementations in practice.

Global frequency and voltage scaling: The most common implementation of DVFS scales voltage and frequency for all cores simultaneously. It requires a single voltage controller. Hence, all cores run at the same frequency, so it cannot be used for differential power delivery.

Individual Frequency and Voltage Scaling: Introduced with the Haswell-EP [15] and AMD Ryzen [4] processors, individual DVFS allows each core to have a different individual voltage and frequency levels. However, this approach increases hardware complexity by requiring on-chip per-core voltage regulators (e.g., Intel's FIVR [11]). In between global and individual frequencies, a processor may also support $1 < N < \#cores$ levels. For example, the Ryzen 1700x allows only 3 unique voltage/frequency combinations across 8 cores, although the frequencies and voltages are configurable [4, 31, 38]. We discuss this limitation and its workaround in the next section. This workaround leads to an optimization problem of determining which three frequencies are optimal for a set of workloads. For finer-grained control processors may allow setting voltage and frequency separately .

Performance states (P-States) The standard way to communicate DVFS states from supervisory software is performance states (P-States), which represent different power/performance levels. This interface is presented to the user via the following API's:

ACPI P-States: This is the industry standardized interface for setting the processor DVFS state [50]. There are 16 possible states starting from P0 being the highest frequency state and higher states having lower operating frequencies.

Model-specific register: Recent processors provide model-specific registers (MSRs) for direct control over voltage and frequency. This can be done using a vendor provided driver,

such as Intel P-State driver in Linux [57] or directly writing frequency and voltage levels to relevant MSR registers for AMD Ryzen devices. The Intel devices provide 100MHz frequency steps and AMD Ryzen provides 25MHz steps.
Limitations of P-States: While designed as a power/performance control mechanism, P-States can have different effects on each application because the power savings of moving to a higher P-State and the resulting loss of performance depends on workload characteristics. The speed of memory and I/O does not change with frequency, so changing P-state has less impact on the performance of memory- and I/O-bound applications. As a result, using P-states to control useful metrics like power consumption or performance requires dynamic adaptation to the workload.

There has been an attempt to reconcile this using a new interface in ACPI known as the Collaborative Processor Performance Control (CPPC) interface [50]. With this mechanism, hardware controls DVFS settings and software provides a range of allowable performance. Intel's implementation of CPPC is called Hardware P-States (HWP) [1]. However, the performance level used by CPPC is specific to the hardware implementation, and hence needs to be tuned for a specific workload and machine.

Opportunistic Scaling Modern processors provide the ability to overclock cores when only a subset of the total cores is active. In these modes, such as Intel's TurboBoost and, AMD's Precision Boost and Extended Frequency Range (XFR), when some cores are idle and provide power/thermal headroom, the remaining cores may run at higher power, and hence higher frequencies [3, 22]. This provides a benefit to sequential code while staying within power limits.

Core Idling (C-States) With sleep states, the processor is forced to idle, which reduces the frequency to zero until the core receives an interrupt. While these are the lowest-power states, they do no computation and take longer to enter and exit (1-200 μsec [46]). At idle the most efficient processors consume power in milliwatt range compared to 10s of watts at maximum frequency. The non-idle state or active state for the processor is denoted as C0. All other C states represent different levels of sleep states for an idle core. Core idling, even when there are jobs waiting to execute, can be useful to provide more power, and hence more performance (even opportunistic scaling) to high priority tasks running on the remaining active cores.

2.2 Power capping

The following mechanisms and interfaces are provided for limiting power consumption.

Running Average Power Limit (RAPL) This interface provides software with the ability to monitor, control, and receive notifications of processor energy and power consumption [22]. To allow for granular power control, Intel divides the processor into multiple power domains such as package, DRAM controller, CPU cores, graphics and uncore [25, 43]. RAPL allows an operator to set a limit on the amount of power a particular domain can use, and the system dynamically adjusts voltage and frequency on fine time scales to stay within the limit. This mechanism has been adopted by other vendors as a viable way to provide deterministic power draw for a processor [9]. When RAPL is implemented with global DVFS, all cores are throttled to the same frequency, even if one core uses much more power than others. As we have limited control over who gets throttled, RAPL alone is unable to provide differential power to applications.

OS Frequency Governors Software heuristics are used to determine the next P-State of system based on the current system usage / load. Examples include *Power Plans* in Windows [37], and *cpufreq* in Linux [7]. These governors may limit peak CPU speeds, or the time spent at peak speed to save power/energy. A common policy is to look at CPU utilization and reduce frequency when the CPU has low utilization. These governors' express the desired OS policy to the hardware as requested P-States. In our experiments we use the *userspace* governor to manually set the P-states from a userspace application.

Thermal Daemon Linux also offers the *thermald* interface to the processor's thermal management features. This interface allows the user to set thermal limits on the system. When triggered these limits can use P-states, RAPL, C-states and even clock cycle gating [22] to reduce power consumption. The thermal daemon selects the set points for the various mechanisms based on provided thermal limits. Depending on the mechanisms enabled to maintain the thermal limits it can have differing effects on application performance. As these mechanisms can be both global (RAPL) or local (clock cycle gating, DVFS), they may be helpful in building a per-application power delivery system.

3 Effectiveness of Power Management Mechanisms

We study the common power management mechanisms (DVFS and RAPL power capping) to assess their impact on application power draw and performance, and to determine their viability for differential power delivery on our test platforms. The test platforms are two recent microprocessors: Intel Xeon-SP 4114 (Skylake) and AMD Ryzen 1700X. Table 1 provides a summary of features provided by each CPU. Note that the study is meant to only evaluate the processor mechanisms, *not* the platforms we chose to evaluate them on. We thus look only at processor and not whole system power. While many past studies have reported on the effectiveness of DVFS, we include results here to motivate per-application polices described in Section 4.

Table 1. Summary of Power Management Features (Mechanisms) Available

Processor	Features
Skylake: Xeon SP 4114	◦ 2 sockets, 10 cores, 20 threads, 192 GB DRAM ◦ 0.8–2.2 GHz + 3 GHz TurboBoost ◦ Per-Core DVFS, 100MHz increments ◦ RAPL power capping (20-85 W) ◦ Platform power measurements
Ryzen: AMD Ryzen 1700X	◦ 8 cores, 16 threads, 16 GB DRAM ◦ 0.4–3.4 GHz + 3.8 GHz XFR2 ◦ Per-core-DVFS, 3 simultaneous P-states, 25 MHz increments ◦ Platform and per-core power measurements

3.1 Methodology and Experimental Setup

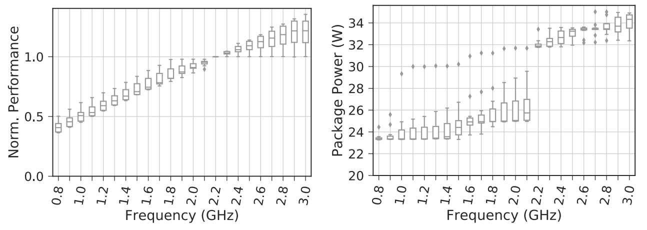
We evaluate power management mechanisms and our policies using a recommended subset of 11 benchmarks (*lbm, cactusBSSN, povray, imagick, cam4, cpugcc, exchange2, deepsjeng, leela, perlbench, omnetpp*) from SPEC2017 [30]. These benchmarks provide a comparative measure of compute intensive performance enabling us to study, in a controlled way, the effects of various power management techniques on applications. The goal for this work is to manage application performance loss when applications are co-located under a power cap. Having benchmarks with steady performance characteristics simplifies analysis, as there are few large phase changes that dramatically change application behavior.

We measure power consumption using RAPL interfaces, which we verified using a Watts Up meter and which have previously been shown to be accurate [26]. We recorded the following variables to aid our analysis of the mechanisms:

- *Package power*: power consumed by the whole platform, including the core and uncore, as reported by RAPL interface.
- *Core power*: power consumed by the cores as reported by RAPL interface, available only on Ryzen
- *Performance*: instruction count per second for the process.
- *Active frequency*: The frequency of operation for the processor when it is in C0 state (Active/Non-idle).

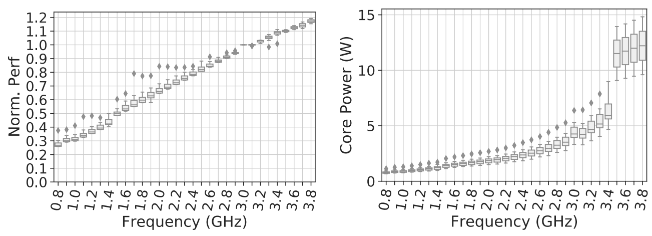
We use the *turbostat* tool to collect these variables once per second [8]. We modify this tool to support monitoring the AMD Ryzen processor.

For Skylake, we directly set the P-states for each core, with each P-State corresponding to a different fixed frequency. For Ryzen, our processor is limited to 3 P-States, though, we can redefine the frequency of these 3 P-states. Thus, we use P-State 0 (P0) for frequencies 3.4–3.8GHz, P1 for frequencies 2.2–3.3 GHz, and P2 for 0.8–2.1 GHz. Each P-state uses the same voltage level (the default one configured by the BIOS) for all frequencies it represents.



(a) Performance normalized to 2.2GHz. **(b)** Average package power.

Figure 2. Effects of DVFS on Skylake for SPEC2017 workloads. Power values are for the package, averaged over application runtime. The line denotes median, box denotes the 1st and 3rd quartiles of the dataset, and the whiskers show the 1st and 99th percentiles of the distribution, with outliers marked as stars.



(a) Performance normalized 3.0GHz. **(b)** Average core power.

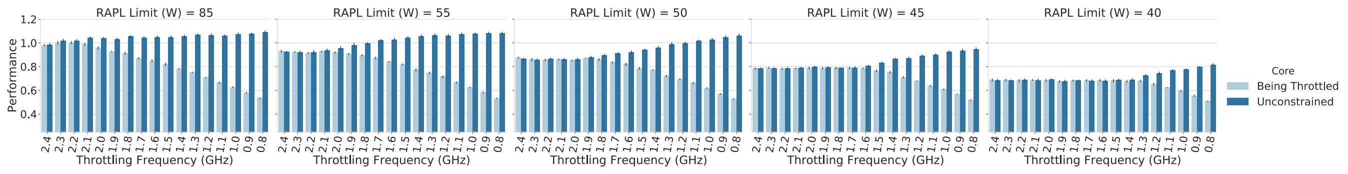
Figure 3. Effects of DVFS on Ryzen for SPEC2017. The line denotes median, box denotes the 1st and 3rd quartiles of the dataset, and the whiskers show the 1st and 99th percentiles of the distribution, with outliers marked as stars.

3.2 Results

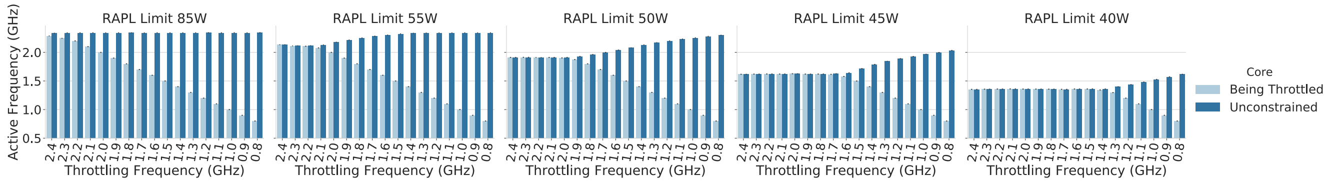
We first conduct a sweep of power management parameters on both platforms to determine the effect of P-states (DVFS) on power and performance. We then evaluate the interaction of per-core DVFS and RAPL power limiting.

DVFS P-states We begin with a study of the effectiveness of DVFS on both Intel and AMD platforms. We analyze DVFS by first setting all cores to the same P-state and then running the benchmark, pinned to an isolated core, to completion. We show normalized runtime and average power over the range of available frequencies in Figure 2 for Skylake and Figure 3 for Ryzen. The performance is normalized to running the application at 3.0 GHz for Ryzen, and 2.2GHz for Skylake.

The effect of DVFS on power and performance varies widely by application. This is observed in the wide range of results across the 11 benchmark programs and has been shown previously (e.g., by Koala [48] and others). On Skylake, two features in particular stand out. First, the results show several power outliers; these applications (*lbm*, *imagick* and *cam4*) make use of Advanced Vector Extension (AVX) instructions that are relatively high power. Second, we see



(a) Performance normalized to 2.5GHz and 85W RAPL limit



(b) Active Frequency

Figure 4. Impact of RAPL on per-core DVFS with *gcc* benchmark. Half the cores are unconstrained at 2.5GHz and the other half are throttled to the frequency on the X axis, while progressively lowering the power limit with RAPL.

that for these applications the performance peaks at a relatively low 1.9, well below the limit of 3.0 GHz. Using AVX instructions reduces the maximum processor frequency [29]. This result demonstrates the necessity of active measurements to determine if higher power states are necessary for a given workload [1]. Finally, we observe that at 2.2 GHz and above, power jumps by about 5 watts. This represents the change from regular operation to TurboBoost modes, which draw substantially higher power.

For Ryzen the results are similar, but with smaller anomalies. Performance does not show any saturation effects, as it increases nearly linearly as frequency increases. Like Skylake, Ryzen shows a jump in power use for SPEC workloads at 3.5 GHz, which when Precision boost and XFR (like TurboBoost) frequency levels take effect.

While we presented results for applications running on a single core, using per-core DVFS and multiple applications or multi-threaded applications resulted in similar power/performance graphs.

RAPL power limits Past work has shown that RAPL limits provide stability, accuracy, and fast settling time [59], so we do not present results showing its basic functionality. As mentioned in Section 1 and shown in Figure 1, with global DVFS, RAPL throttles all cores to the same frequency, so cores using less than their fair share of power may be throttled due to unrelated high-power applications.

We evaluate the ability of RAPL with per-core DVFS as a possible solution for differential power delivery on the Skylake platform, which supports both (Ryzen lacks RAPL limits). We run copies of the same application (*gcc*) on all available cores. We configure half the cores to be unconstrained (running at the maximum of 2.5GHz) while the remainder we set to a progressively lower frequency. We repeat this with a range of power limits from 85W (the processor's TDP) to 40W.

The results are presented in Figure 4. We see two salient features from these results. First, Figure 4(b) shows that RAPL finds a global maximum frequency to keep the system under the power limit. Second, Figure 4(a) shows that the power saved by throttled cores is used by the unconstrained cores to run faster. For example, at 50W, when the throttled core runs at 0.8 MHz, the unconstrained core performance improves from 14% below its performance at 2.5GHz to 6% above. Thus, *per-core DVFS provides an effective mechanism to achieve differential power delivery.*

However, we also observe the policy used to apply power limits when cores have different frequencies: from the frequencies of the throttled core in Figure 4(b), it is apparent that *RAPL only reduces the frequency of the unconstrained core.* This effect is seen once we start throttling below the global frequency determined by RAPL. This policy is effective for saving power, as it takes power from the most power-hungry cores.

However, on a system with mixed-priority workloads, this policy may be misguided: it may be preferable to take power from lower priority workloads, even if they are already throttled, and leave power to isolate and maintain the performance of high-priority workload as much as possible.

Unfair throttling We demonstrate the central problem with not having policy control when using RAPL as a power management mechanism by running two applications on 10 cores, one considered high priority and one low priority. The low priority application (*websearch* [14, 42]) has a low power demand as it consumes only 44 watts with 9 active cores at 3GHz, while the low-priority application (*cpuburn* [40]) consumes 32 watts with only 1 active core. We run these two applications simultaneously under a power limit.

We present the performance (normalized 90th percentile latencies for 300 users) for *websearch* with and without co-location run under a RAPL limit in Figure 5. We observe a

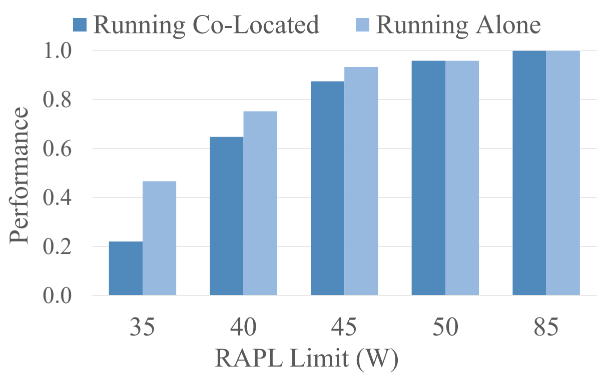


Figure 5. Effect of co-location under RAPL. We have a latency sensitive application (*cloudsuite/websearch*) co-located with a power virus (*cpuburn*). The figure shows the 90th percentile latencies for *websearch* with and without co-location, running under progressively lower RAPL limits, with frequency set at 3GHz.

dramatic decrease in performance (less than 50% of when running alone) as a result of the presence of a single low-priority application, especially when running under lower power budget (<40 W). This represents a challenge for any platform that multiplexes high and low priority applications on the same hardware at the same time: *power-intensive low-priority applications may trigger power limits, which hurt the performance of high-priority applications*. Indeed, this has been documented in internet service data centers [33].

With this result in mind, in the following section we develop and evaluate simple policies to control an ideal platform that allows us to perform per-application power management and isolation.

4 Policy Design

In this section we describe a set of policies that allow an operator to request differential power delivery among co-located applications. The ultimate goal for the policies is to treat power like other system resources, such as memory or CPU time, that are scheduled or allocated by the operating system. In particular, we focus on two classes of policies. *Priority policies* try to ensure that higher-priority applications run at maximum performance and use residual power for lower priority applications. *Proportional-share policies* divide power between applications according to operator-configured shares. Priority policies, while simple to implement, are rigid in treating mixed priority scenarios and can lead to starvation. Proportional share policies, common for CPU [55] and resource allocation [54], provide more flexible control over power distribution.

In this work we focus on isolation and sharing strategies for power. We discuss but do not consider time-sharing (applications running on the same core) in detail, as it requires modifying the scheduler to be power aware, while we focus on power management separately from scheduling. We also do not consider real-time policies, such as those that provide a performance or power guarantee to applications. We focus on policies for applications running concurrently on different cores (space-sharing).

4.1 Priority Policy

We classify applications as *high priority* (HP) or *low priority* (LP) based on their relative importance. In strict priority policies, low-priority applications only run after all high-priority applications' needs are satisfied. In contrast, with proportional-share policies, low-priority applications run but receive a lower share of power than high-priority applications.

When designing a priority policy, we consider the *power demand* of an application: in a given P-state, does an application use more power than other running applications, or less power? We term those using more power *high demand* (HD) and those using less power *low demand* (LD). Combining priority and power demand, we have four categories of applications we name HDHP, HDLP, LDHP, LDLP. The central scenario where differential power delivery is most valuable is when there are low-demand/high-priority (LDHP) applications running simultaneously with high-demand applications. A high-demand (HDLP) application may trigger power limits that unfairly harm the low demand yet high priority (LDHP) application.

Strict priority policies only give resources to low-priority applications once high-priority application have been fully satisfied. In the context of power, this means that high-priority applications should be able to use power up to their demand, and only left-over power is granted to low-priority applications. When applications have equal priority, this scenario devolves into a proportional share policy with equal shares, which we consider in the next section.

The fundamental priority policy here sets the P-state for HP application(s) to the maximum possible under the power limit, and the P-state for LP application(s) to the maximum possible with the residual power. In the absence of a separate proportional share policy, all HP and all LP applications run at the same P-states. We note that this policy may cause starvation at lower power limits, in cases where there may not be enough power left to run LP applications at their minimum P-state. As an alternative, the policy can be modified to first allocate the minimum required power to all cores to execute before allocating additional power for high-priority application to run at maximum performance. Since this is a choice for the implementation, we discuss our choice in the implementation section.

Simple priority policies, while easy to implement, are rigid in treating mixed priority scenarios and can lead to under-utilization, or even starvation. To overcome this, we propose *proportional share* policies that provide more flexible control over power distribution.

4.2 Proportional share policies

Proportional share policies distribute a resource according to the relative shares of active applications: if an application with 3 shares runs concurrently with an application having 1 share, the first application receives 3/4ths of the resource and the second receives 1/4th. A central question, though, is *what is the resource* being distributed?

While power itself is an obvious answer, we identify three possible resources to share proportionally.

1. *Power*: The power draw of each application should be proportional to its shares. This requires monitoring power demand of running applications, adjusting P-states locally at each core to hit power targets, and only adjusting P-states for all cores when there is excess power to redistribute.
2. *Frequency*: The frequency at which applications run should be proportional to the shares they hold. As frequency correlates highly with performance, this is a simple form of proportional performance. It requires monitoring global power draw and adjusting frequencies for all cores to redistribute power when it deviates from the power limit.
3. *Performance loss*: The performance of applications, relative to running in isolation without a power limit, should be proportional to shares, so applications with more shares suffer less performance loss. This requires monitoring application performance and total power draw and adjusting frequencies on all cores to redistribute performance when total power deviates from the limit.

These three resource options for proportional shares, have different properties. Power shares are attractive as they relate directly to the resource being allocated and allow local control with infrequent global redistribution. However, they may require knowledge of application power demand to set shares (e.g., understanding how much power is needed for adequate performance), about which operators may be unfamiliar. Frequency shares are simple to reason about, require little hardware support (just global power measurements and per-core DVFS) and no prior application knowledge is required. However, it may provide non-intuitive behavior. Finally, performance shares control what operators care most about, performance, but require an accurate/meaningful measure of application performance.

4.3 Single-core Sharing Policy

On a single core applications time share the CPU. To control the fraction of the core for an application, the user can specify its CPU shares using the cgroups cpusets feature [13], or by changing the Linux priorities of the application. In this case the applications can present themselves in the following combinations:

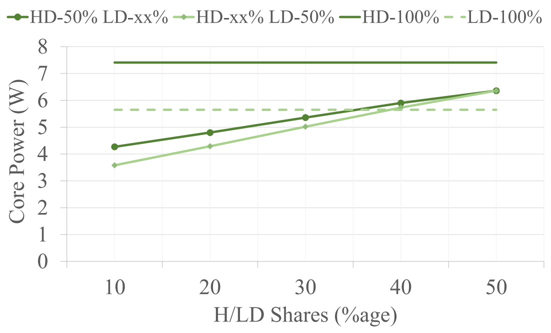


Figure 6. Time-shared power consumption for *cactusBSSN (HD)/*gcc (LD) applications running on the same core. We present both cases, when one app (HD or LD) has been allocated 50% of the core and the shares for the other app (LD or HD) are varied from 10% to 50%. Also shown is the power consumed when either of the apps are given 100% share of the CPU (i.e run alone) at 3.4GHz on the Ryzen Platform.

1. *Equal demands, mixed shares, mixed priorities*: In this case the power will be the similar for both applications. For this scenario the policy sets core P-state to highest level that can run either application and stay within power limit.
2. *Mixed demands, equal shares, same priorities*: With equal shares, under a power limit the processor frequency must be reduced to run high-demand applications, which throttles low-demand applications unnecessarily. CPU scheduling can be modified to give low-demand applications more runtime, by dynamically adjusting their CPU shares at runtime to compensate for CPU throttling.
3. *Mixed demands, mixed shares, mixed priorities*: The core should be set to run the high-priority application at the highest level possible within the power limit. If one application is HDHP, the LDLP application runs at the same frequency, which is slower than if it ran alone. If one application is LDHP, the core runs at its maximum frequency and the HDLP application does not run at all (if it exceeds the power limit).

To demonstrate that this policy can be implemented using existing mechanisms, we ran HD and LD applications as docker containers and varied their CPU shares using docker daemon [36]. We fixed one of the HD/LD applications at 50% CPU share and varied the share for the other from 10% to 50%. The observations for power consumption are presented in Figure 6. We note that varying CPU shares changed the amount of time the apps were resident on the core.

We observe that the change in time resident on core results in the average power consumption for the core to increase or decrease proportionally. A closer look reveals that the power drawn by the core is a time-weighted sum of the individual application power draws. Thus, a power mechanism for time sharing can achieve the desired power and performance by varying both processor frequency and per-application CPU share.

4.4 Discussion

We note that these policies do not consider applications with performance or power that saturates, such as applications that perform no faster when run at higher frequencies (see Figure 2(a)). To handle this case, both priority and proportional-share policies can be modified to try to run applications at the highest useful frequency rather than the highest possible frequency. Hardware support such as Intel's HWP [1] can help identify this point. Furthermore, these policies do not address the situation where there is not enough power to run all applications in a class, such as all low-priority applications with the power remaining after high-priority applications are satisfied.

We also note that these simple policies can lead to starvation under space sharing even when a subset of applications could still run. For example, with a priority policy, there may not enough power for all low-power applications to run, but there is enough for a subset. In this case, the policy should disable cores (put them in a sleep state) and let the OS scheduler time-slice applications on the remaining cores.

5 Policy Implementation

To better understand how our proposed policies, affect applications, we designed a userspace daemon to implement them. The daemon dynamically sets the target P-state configuration based on power limit, policy (shares or priority) and power feedback. It takes a list of programs as input with their priority and shares. This information is used to select the initial set of applications to run and the initial state of the system. Applications are pinned to cores, and their priorities or shares are used to select an initial P-state for each core. The daemon then runs a monitoring loop. In every loop iteration (1 second in our implementation), it reads processor statistics, including power (per-core or per-package), performance (retired instruction count), and actual frequency. Based on the measured values, the daemon may change P-states for a subset of cores. If the daemon detects an application/core is using more or less of a resource than it was allocated, it can either increase the frequency of the core or redistribute the resource to other cores, called *re-distribution.*

5.1 Implementing priorities

Under all policies, the daemon monitors package power and adjusts P-states if total power is above or below the target. For priority policies, the daemon starts the HP applications at the highest P-state. If the total power is above the target, the daemon lowers the P-state of all HP applications until they are within the budget. This uses one of the proportional share policies described below. If, on the other hand, there is excess power after all HP applications are running at maximum speed, the daemon starts LP applications at the slowest P-State. It then increases their P-state (according to a proportional share policy) until the system reaches the power limit. Thus, at low power limits with HDHP applications, LP applications may suffer starvation if not all can be started at the slowest P-state.

We note that when only a few applications are running, the processor may enable TurboBoost (Intel) or XFR (AMD) and run above normal frequencies. When all cores are used through, these higher frequencies are not available. Thus, a policy decision must be made as to whether to starve LP applications to enable HP applications to use these higher frequencies, or to allow the LP applications to run at the lowest frequency without hurting the HP application. In our implementation we starve the LP applications.

5.2 Implementing shares

For proportional-share policies, the daemon starts all applications with an initial estimate of the correct P-states, and then adjusts the voltage and frequency to satisfy both share proportions and the power limit. We describe below the exact control loop for each resource.

When there is excess power, we use a min-funding revocation policy [54] to distribute the excess across applications that are not running at the maximum frequency. We implement this by removing saturated applications from the mix, then re-running the distribution algorithm across the remaining resources and remaining applications.

We note that unlike memory or CPU, there is a low dynamic range for shares of power: as shown in Section 3, frequency only varies by a factor of 3–4, core power by a factor of 12-14, and performance by a factor of 4. Consequently, not all share ratios are possible. Assigning a share ratio of 99:1 leads to two outcomes: either the low-share application starves when the high-share application is running, or it uses a larger fraction of resources than its share. If we choose starvation, we get perfect isolation, but may cause the resource to be underutilized when the high-share application cannot use all the available power. Letting the low-share application use a larger fraction improves resource utilization but breaks isolation. In our implementation, we only allow starvation with priority policies and with shares we allow all applications to run at least at the minimum frequency.

All share mechanisms are implemented using three functions: (i) an initial distribution function to run when starting applications, (ii) a redistribution function to run when power is above or below target, and (iii) a translation function that converts units of the managed resource to frequencies that can be programmed into the CPU. The *initial distribution* finds the initial resource allocations for the given set of applications. The *redistribution function* is similar to the initial distribution function, but has the additional responsibility of applying the min-funding revocation policy to handle excesses and shortages of the shared resource and of identifying saturation. Saturation means that a core has reached the maximum (or minimum) value possible for the shared resource and hence usefully use more (or give up less) of the resource. The *translation function* takes the result of the redistribution function (changes to resource allocations for cores) and translates them into target frequencies.

Since these functions can vary according to the policy, we discuss them in detail below.

Power shares For this policy we care that applications share power proportionally. This requires that we have per-application power feedback for our control loop. We implement this policy only on Ryzen platform as it fulfills this requirement.

The *initial distribution function* for this policy distributes the power limit among the applications based on their share ratios. The result of this distribution is a set of per-application limits.

The *redistribution function* updates per-application limits by distributing the difference in current power and the power limit among non-saturated cores.

For the *translation function*, we use a power model to predict the initial distribution of frequencies, and then every iteration adjusts the frequency values based on the power feedback from each core and the calculated limits. The power model is simple linear equation that converts the power range to the frequency range. Since we dynamically adjust the values later, modeling errors do not affect steady state behavior.

Frequency Shares In this policy applications share frequency proportionally. This requires per-application frequency measurements, which are supported on both Ryzen and Skylake.

Since the shared entity is frequency and our limits are specified in power, this policy requires translating the power limit into a frequency limit, the opposite of what was needed for power shares. The *translation function* uses a conversion factor α as a function of the change in power (*PowerDelta*):

$$\alpha = (PowerDelta/MaxPower)$$
$$FrequencyDelta = (\alpha * MaxFrequency* \\ NumAvailableCores)$$

This naïve model allows the daemon to estimate how much frequency (cycles) must be distributed or withdrawn from a target to change power. The *FrequencyDelta* is applied proportionally to all applications, resulting in a new set of per-application frequency limits. While this model is simplistic, the error becomes smaller when the system is near the target power.

The *initial distribution function* sets the highest-share application to the maximum frequency and remaining applications to their proportions of the maximum frequency. The *redistribution function* computes the difference in power used to the target, converts it to frequency, and distributes the frequency among non-saturated cores. The *translation function* converts the target frequencies into valid (quantized) frequencies for the platform.

Performance Shares For this policy we care that applications share performance proportionally. It requires per-application performance feedback for our control loop. We use instructions-per-second (IPS) as a proxy for performance, as our workloads are single-threaded. For multithreaded workloads with lock contention, where spinlocks may artificially inflate instruction counts, hardware mechanisms such as Intel's HWP with its abstract performance metric [1] may be a better choice. IPS is available on both Ryzen and Skylake platforms.

As a baseline, we use the performance of an application running alone at maximum frequency (measured offline). We normalize IPS to the baseline to get the performance number used for power distribution. We convert the power limit into a performance limit, using α and then multiplying it to the maximum core performance and the number of cores we can allocate.

$$PerformanceDelta = (\alpha * MaxPerformance* \\ NumAvailableCores)$$

The *initial distribution function* distributes this performance limit among the applications based on their share ratios. The result of this distribution is a set of per-application performance limits. The *redistribution function* updates these per-application limits by first converting the difference in current power and the power limit into a performance value and the distributing it among non-saturated cores.

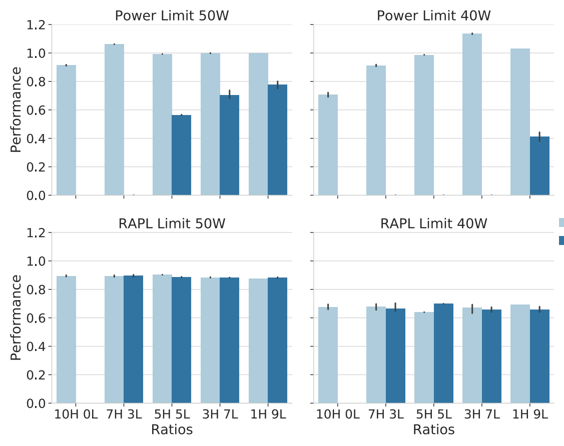
The *translation function* uses α to translate power into performance, and performance into an updated set of frequencies. The performance model here is similar to the power model described above.

Ryzen details The Ryzen platform only supports use of 3 unique P-states concurrently, so we built an additional selection utility that dynamically reduces the target frequencies to three valid P-States.

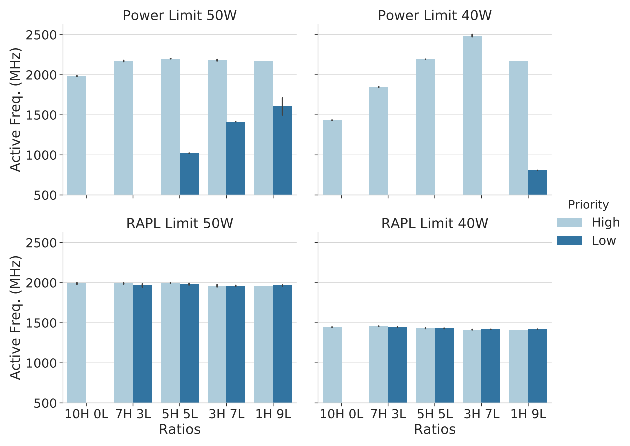
Our control daemon is not designed for production use, but instead to demonstrate the effects of different power management policies. The policy should be implemented in hardware, similar to RAPL, to provide a low sampling overhead and have a fast response to changing workloads, and workload characteristics.

6 Policy Evaluation

We present two sets of evaluation results. First, using carefully constructed workloads from SPEC CPU2017, we *demonstrate* the capability of our policies and, the Skylake and Ryzen platforms to implement differential power delivery. We chose *cactusBSSN* as a HD application and *leela* as a LD application, to show how our policies perform across a variety of settings. Second, we *measure* the ability of our policies



(a) Measured performance normalized to standalone performance at 85W.



(b) Active frequency reported for the two priority levels

Figure 7. Priority experiments on Skylake. The Upper graph in each pair is the priority policy and the lower is RAPL. The number of HP application ranges from 10 to 1 and is indicated in the ratios. The values in each bar is averaged over all active applications.

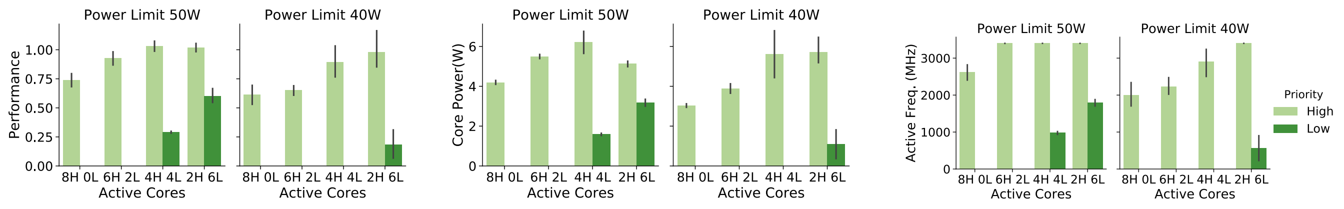


Figure 8. Priority Policy experiments on Ryzen. The number of high-priority application ranges from 8 to 2. Performance is normalized to standalone performance at 85W.

to enforce useful properties by running them with specified shares and random mixes of applications. These results evaluate how effective our policies are in realistic settings at providing differential power delivery and performance isolation.

We evaluated both policy types on both Skylake and Ryzen with the exception of power shares, which we only ran on Ryzen. For brevity, we do not show all the results but they are available at http://research.cs.wisc.edu/sonar/power.html, both in tabular form and as graphs. For the curious readers, we also provide our scripts and the userspace daemon.

6.1 Priority Experiments

For the priority experiments, we run as many applications as there are cores on a processor and vary how many are high or low priority. In each case we have an equal number of applications that are LD or HD.

Figure 7 shows the functioning of the priority policy on Xeon for two different power limits (50W, and 40W). We execute a mix of high-priority (HP) and low-priority (LP) applications, which are split between *cactusBSSN* (HD) and *leela* (LD). For LP jobs, there is one more *leela* and for HP jobs there is one more *cactusBSSN*. The workload mixes used for Skylake are shown in Table 2

Table 2. Workload mixes for Skylake priority experiments

Mix	*cactusBSSN*-HP	*leela*-HP	*cactusBSSN*-LP	*leela*-LP
10H 0L	5	5		
7H 3L	4	3	1	2
5H 5L	5			5
3H 7L	2	1	3	4
1H 9L	1		4	5

The performance results at the top of Figure 7a show that at lower limits, the priority policy results in starvation when there are too many high-priority applications. As discussed in Section 4, an alternate policy would be to throttle high-priority applications to allow low-priority applications to run. At 85W there is enough power to run all applications at full speed. At 50W, there is only enough power to run LP applications when there are 5 or fewer HP applications. The results also show the policy can leverage opportunistic scaling. At 40W, when there are only three HP applications, they run faster than at 85W. This occurs because there is not enough power for all seven LP applications to run; instead that power is used to boost the speed of the HP applications.

We note that an alternate policy would be to only disable some of the 7 cores running LP applications, and time slice them on the remaining cores. Finally, we see at 50W that as there are fewer HP applications and their performance saturates, the LP applications take advantage of the extra power to run faster.

In contrast, with RAPL (Figure 7 bottom) there is no distinction between HP and LP applications, so under power limits both applications suffer similar reduction in frequency and performance.

Figure 8 shows the results for the priority policy on Ryzen, which is almost identical to those on Skylake. Here, we show core power as well (middle figure). There are no RAPL results, as the mechanism is not documented on the platform.

Here the workload mix has four variations of having mixed (2HP-6LP, 6H-2L cases) and similar (8HP, 4HP-4LP) demand workloads. When we limit the system to 50W, LP jobs can only run when there are 4 or fewer HP jobs. At 40W, they only run when there are 2 HP jobs. We note that there is a slight reduction in core power at 50W going from 4H4L to 2H6L; this occurs because the 4H are all HD, while the 2H is a mixed workload of HD and LD.

6.2 Proportional Share Experiments

For proportional-share experiments, we choose two share levels and assign half the cores to one share level running *leela* (LD) and half to another running *cactusBSSN* (HD). We visualize the data two different ways.

Figure 9 shows the results for the proportional share policies on Skylake. With native RAPL, all applications run at almost the same frequency and the HD application achieves 8% higher performance. We do not show power shares, as they require per-core power measurements, which the Skylake platform does not provide.

The Skylake results show two key results. First, there is a low dynamic range for resource allocation, processor only supports 800MHz–2200MHz operation. Thus, at 90/10 share ratios, the low-share application receives more than its share of frequency or power. Second, we see that frequency and performance shares have very similar results. This indicates that a simple frequency share policy, which is more stable (it does not change with program phase) may work as well as a more complicated performance share policy.

Figure 10 shows similar results for the proportional share policies on Ryzen. This visualization shows the relative resource use by each application across frequency, performance, and power shares. Here we see similar results to Skylake. In general, the daemon is able to accurately share resources for the 30/70, 50/50, and 70/30 cases, but cannot achieve less than 20% resource usage. This occurs due to the high minimum frequency (800 MHz). We also see that frequency shares on Ryzen provide the most accurate control over performance. In contrast, performance shares often under- or over-shoot the target. This occurs because frequency is stable while running, while performance is measured as IPS relative to the long-term average IPS for the program. Small phase changes can affect performance, leading to control operations to rebalance power. This can be observed in 30/70 case for performance shares at the 40W limit. Finally, we can also observe that power shares in general provide poor performance isolation due to the differing power demands and varied behavior of applications.

6.3 Random Experiments

The preceding experiments use hand-selected high demand and low demand applications to demonstrate the behavior of the policies. For more generalizable results, we perform similar experiments using randomly selected subsets of SPEC2017 workloads (using numbergenerator.org). We create two sets, A and B, that are listed in Table 3. For Skylake, we run two copies of each of the 5 applications, with both copies given the same share. The share levels for Skylake platform are {20, 40, 60, 80 and 100}.

Table 3. Applications for random experiments.

App. #	0	1	2	3	4
Skylake A	deepsjeng	perlbench	cactusBSSN	exchange	gcc
Skylake B	deepsjeng	omentpp	perlbench	cam4	lbm

Figure 11 shows the results for random experiments for the proportional share policies on Skylake. For the A set of applications (left half of each graph), we see the expected results: as shares increase, power and performance increase. The power and frequency shares policies mostly achieve the same results. Application A3 (*exchange*) performs worse and Application A1 (*perlbench*) better than expected with performance shares due to their higher (lower) sensitivity to frequency. At 40W, we see little change in performance for applications A1–A3 with the frequency policy. This occurs because the dynamic range of frequencies is small here (100s of MHz), so there is not enough range to achieve proportionality. For the B set application on the right, we see dramatically different results. As seen at 85W, application B3 and B4 cannot run at full frequency; this is because both use AVX instructions. At 50W, the frequency shares behave similar to as expected, with increasing performance as share increases, even though frequencies saturate.

6.4 Latency-Sensitive Experiments

In this experiment we repeat the unfair throttling experiment presented in Section 3 with our power policies. We run two applications: a latency-sensitive multithreaded application (*websearch* [14, 42]) colocated with a power virus (*cpuburn* [40]). *Websearch* occupies 9 of the 10 cores and is considered high priority. *Cpuburn* occupies the 1 remaining core and represents an opportunistic low priority application. We run these applications under progressively decreasing

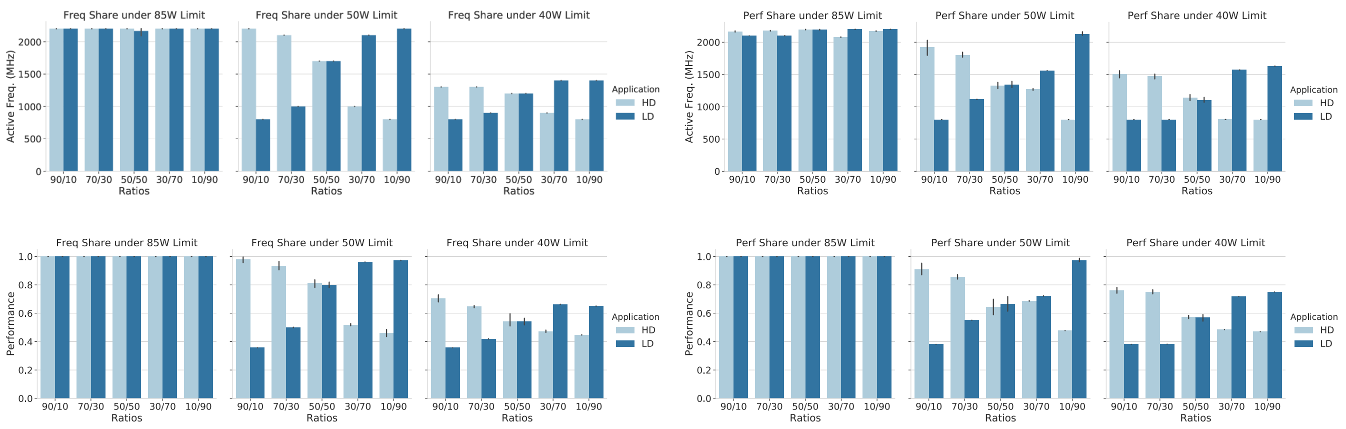


Figure 9. Skylake proportional share policy experiments for Skylake running *leela* (LD) and *cactusBSSN* (HD). The left figures use frequency shares and the right figures use performance shares.

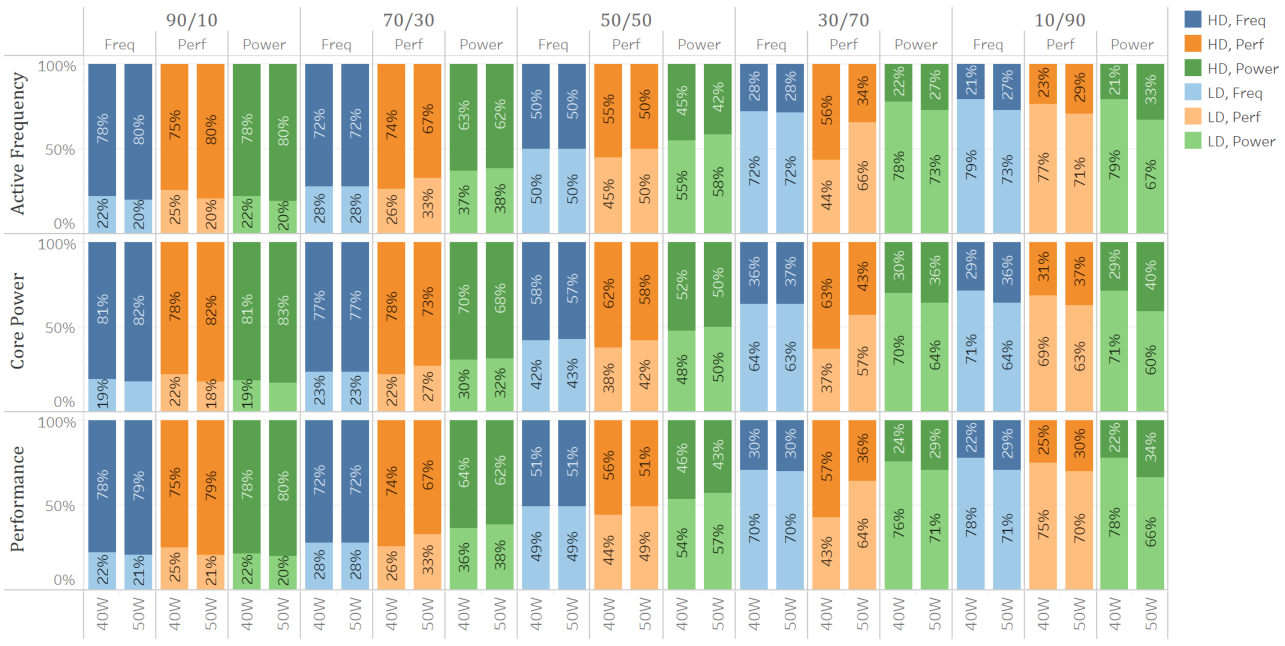


Figure 10. Ryzen proportional share policy experiments. The figure shows percent of total resource used by each application for frequency, performance, and power shares. Each pair of bars shows results for 40W and 50W limits. Each row represents the measured use of a different resource.

power limits on the Skylake platform with the proportional frequency and proportional performance sharing policies. The *websearch* application was loaded with 300 users, doing search transactions for 600 sec. We present our findings with the policies, RAPL and when *websearch* runs alone in figure 12. We ran the experiments multiple times (5) and present the average.

We ran multiple share configurations, but we report on 90/10 share ratio, with each core running *websearch* receiving 90 shares, and the one core running the power virus receiving 10 shares. For this configuration we found that our policy was able to improve performance in all cases, even reaching performance comparable to *websearch* running alone in some cases. For the most power limited cases of 40W and 35W, our policies reduce the performance loss by 10%. As we found the results to vary slightly between runs, the variance in some cases causes our policy looks better than when running alone.

Another thing to note is the reason we are limited to the 10% improvement is because of low dynamic range of frequencies available. This can be seen in figure 13. Note using performance shares (not shown here) provided similar improvements in performance over RAPL.

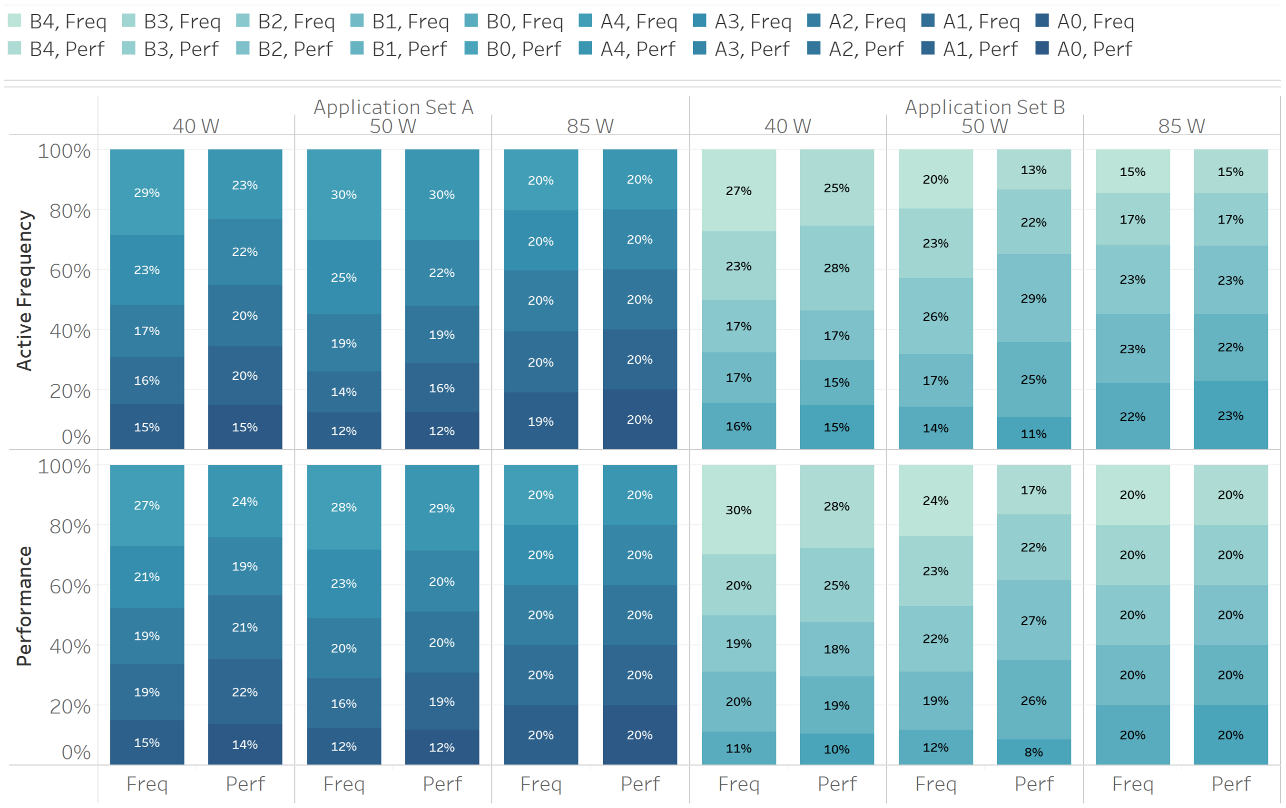


Figure 11. Skylake proportional share policy random experiments for frequency and performance shares at 40W, 50W and 85W. The figure shows percent of total resource used by each application for a particular set of frequency, and performance shares (type indicated below each bar). Each row represents the measured use of a different resource. The share ratio used for all columns is [AB]4 : [AB]3 : [AB]2 : [AB]1 = 100 : 75 : 50 : 25.

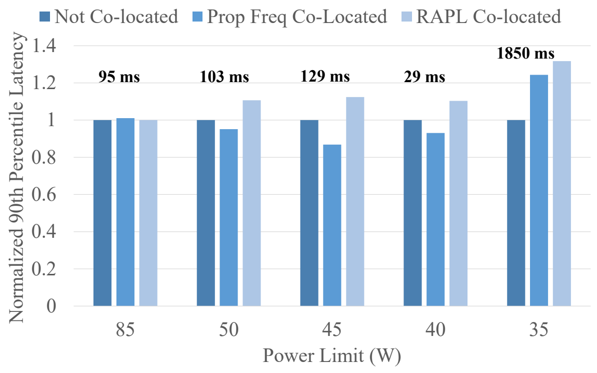


Figure 12. The fgure presents the effect of our policies and RAPL, on 90th percentile latencies, relative to when *web-search* is run alone. The baseline value is noted above the bars).

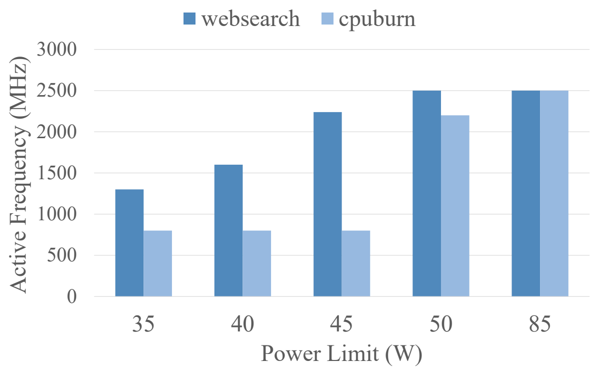


Figure 13. Active frequency measurements for latency sensitive experiments when run with our proportional frequency policy.

7 Related Work

There are many prior works on power management, and we focus on the most relevant categories.

Per-core DVFS for scheduling power: Bircher et al. [6] provide a performance analysis of P-States and C-states on the Intel Nehalem and AMD 10h Family. Vega et al. [52] describe knobs for power management on Power 7 processors targeting efficiency. Isci, et al. [23] provide polices for a multicore system with a global manager for per-core DVFS. The paper assumes presence of global management layer in between the software and the hardware mechanism. They present priority-based policies to follow power limits for this configuration. Lo et al. in their work Heracles [33] discuss the isolation problems that arise due to collocating tasks and motivates our study. Adrenaline [19] talk about polices to utilize a special hardware with fast per-core voltage and frequency switching to reduce tail latencies in warehouse scale workload. Hipster [41] provides a QoS aware energy management system for ARM's big-little cores. In contrast, our work explores currently available power management techniques on x86 platforms common in data centers, and focuses on differential power delivery to applications, not energy efficiency.

Cluster-level methods: There has been much work on power management at the cluster level, which is complementary to our work. No Power work from HP Labs [44] introduces the problem of power management in a cluster and advocates for coordinated systems level solution. They provide an overview of the main strategies for power management at all levels of a cluster hierarchy. Following this work, there have been a whole host of papers talking about power management in a data center [12, 18, 19, 27, 28, 33, 53, 56]. The common thread is that they all use node-level primitives to keep application guarantees. We show that using the most common of these methods (RAPL) is not enough to follow SLA's. We provide a better approach to do node-level power management with policies that use existing mechanisms to achieve operator goals.

DVFS studies: Rubik [24] and Adrenaline [19] focuses on reducing tail latencies by using fine grained DVFS. We try to understand the interactions between various hardware power management features to provide policies for better utilization of the system under a power cap. Marathe et al. performed an empirical study of performance and energy efficiency variation in Intel Processors when put under different DVFS levels with similar utilizations [34]. We repeat some of their studies as a baseline but focus on differential power delivery policies.

Energy and thermal management: Several systems focus on energy management targeting mobile or energy-constrained environments. Cinder[45], ECOSytem[58], and Power Containers[47] manage energy consumption. Other works [17] prevent thermal interference among applications. Our work focuses on the similar goal of promoting power as a primary resource and controlling the power use of every application. Since the focus of these works is energy, they allow long-running applications to accumulate energy over time, while our policies focus on power draw at all times.

Managing power for heterogeneous systems: Scheduling power for heterogenous systems is complementary to our idea of per-application power delivery. The problem has two parts: first finding the most efficient cores for the current application or thread and then deciding on the right settings for throttling under a power constraint. Liu et. al. provides a formal definition for the mapping problem [32]. Craynest et. al provide a performance-impact-based method to derive schedules for such systems [51]. Muthukaruppan et. al. provides the first combined solution focusing on power management [39]. Sozzo et. al. presents a similar approach [49]. These works limit themselves to developing better control mechanisms and have limited policies. Our proposed can be extended to include such mechanisms.

8 Conclusions

Recent processors have the ability to delivery different power levels to each core, but policies to leverage this capability are not yet available. We propose power shares, frequency shares, and performance shares as three alternatives to the current policy of restricting power via an upper limit on frequency. Through experiments we show that all three can delivery per-application power, but frequency shares were the most stable and provided the best performance isolation. The key difference between frequency, power, and performance based policies is that one rewards low power use (power proportionality) while others reward efficient processor use (performance and frequency proportionality).

Another consideration is game-ability: an application can vary its instruction mix to change its measured resource usage. For performance, applications can manipulate their IPS value with NOP instructions, and their power consumption with extra floating-point or vector instructions. Generally, a sound policy is to ensure that any gaming steps an application takes has an overall larger negative impact on their performance than any benefit they might receive from a power allocation policy.

Acknowledgments

This work is supported in part by National Science Foundation (NSF) grants CNS-1617824 and CNS-1533885. We would like to thank our shepherd Christos Kozyrakis and the anonymous reviewers for their invaluable feedback. Swift has a significant financial interest in Microsoft.

References

[1] Kristen Accardi. 2015. Balancing Power and Performance in the Linux Kernel. https://events.static.linuxfound.org/sites/events/files/slides/LinuxConEurope_2015.pdf.

[2] AMD Inc. 2000. AMD PowerNow! Technology. https://support.amd.com/TechDocs/24404a.pdf.

[3] AMD Inc. 2017. AMD SenseMI Technology. https://www.amd.com/en/technologies/sense-mi.

[4] AMD Inc. 2017. Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processors.

[5] L. A. Barroso and U. Hölzle. 2007. The Case for Energy-Proportional Computing. *Computer* 40, 12 (Dec 2007), 33–37. https://doi.org/10.1109/MC.2007.443

[6] W. Lloyd Bircher and Lizy K. John. 2008. Analysis of Dynamic Power Management on Multi-core Processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*. 327–338.

[7] Dominik Brodowski, Nico Golde, Rafael J. Wysocki, and Viresh Kumar. 2016. CPU frequency and voltage scaling code in the Linux(TM) kernel.

https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[8] Len Brown. 2018. turbostat man page. https://www.mankier.com/8/turbostat.

[9] Martha Broyles, Christopher J. Cain, Todd Rosedahl, and Guillermo J. Silva. 2015. IBM EnergyScale for POWER8 Processor-Based Systems.

[10] Martha Broyles, Chris Francois, Andrew Geissler, Gordon Grout, Michael Hollinger, Todd Rosedahl, Guillermo J. Silva, Mark Vanderwiel, Jeff Van Heuklon, and Brian Veale. 2011. IBM EnergyScale for POWER7 Processor-Based Systems. https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=POW03039USEN.

[11] E A Burton, G Schrom, F Paillet, J Douglas, W J Lambert, K Radhakrishnan, and M J Hill. 2014. FIVR – Fully integrated voltage regulators on 4th generation Intel Core SoCs. *2014 IEEE Applied Power Electronics Conference and Exposition* (2014), 432–439.

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. 127–144.

[13] Simon Derr. 2004. Linux Kernel documentation on cgroup cpusets. https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt

[14] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[15] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. 2015. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 896–904.

[16] James Hamilton. 2008. Cost of Power in Large-Scale Data Centers. https://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers/.

[17] J. Hasan, A. Jalote, T. N. Vijaykumar, and C. E. Brodley. 2005. Heat stroke: power-density-based denial of service in SMT. In *11th International Symposium on High-Performance Computer Architecture*. 166–177.

[18] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. 2018. SmoothOperator: Reducing Power Fragmentation and Improving Power Utilization in Large-scale Datacenters. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 535–548.

[19] C. H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture*. 271–282.

[20] Intel Corp. 2004. Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor. http://download.intel.com/design/network/papers/30117401.pdf.

[21] Intel Corp. 2009. Power Management in Intel Architecture Servers. https://www.intel.com/content/dam/support/us/en/documents/motherboards/server/sb/power_management_of_intel_architecture_servers.pdf.

[22] Intel Corp. 2018. Intel 64 and IA-32 architectures software developer's manual. https://software.intel.com/en-us/articles/intel-sdm.

[23] 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* (2006), 347–358.

[24] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast Analytical Power Management for Latency-critical Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*. 598–610.

[25] kernel.org. [n. d.]. Power Capping Framework. https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt. Retrieved 4/1/2018.

[26] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3, 2, Article 9 (March 2018).

[27] Vasileios Kontorinis, Liuyi Eric Zhang, Baris Aksanli, Jack Sampson, Houman Homayoun, Eddie Pettis, Dean M. Tullsen, and Tajana Simunic Rosing. 2012. Managing distributed UPS energy for effective power capping in data centers. *Proceedings - International Symposium on Computer Architecture* 00, c (2012), 488–499.

[28] Jaewon Lee, Changkyu Kim, Kun Lin, Liqun Cheng, Rama Govindaraju, and Jangwoo Kim. 2018. WSMeter: A Performance Evaluation Methodology for Google's Production Warehouse-Scale Computers. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 549–563.

[29] Gregory Lento. 2014. Optimizing Performance with Intel Advanced Vector Extensions. https://computing.llnl.gov/tutorials/linux_clusters/intelAVXperformanceWhitePaper.pdf.

[30] A. Limaye and T. Adegbija. 2018. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software*. 149–158.

[31] Linus Tech Tips. 2017. AMD Extended Frequency Range (XFR) - Explained. https://linustechtips.com/main/topic/850358-amd-extended-frequency-range-xfr-explained.

[32] G. Liu, J. Park, and D. Marculescu. 2013. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *2013 IEEE 31st International Conference on Computer Design*. 54–61.

[33] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42th Annual International Symposium on Computer Architecture.*

[34] Aniruddha Marathe, Yijia Zhang, Grayson Blanks, Nirmal Kumbhare, Abdulla Ghaleb, and Barry Rountree. 2017. An empirical survey of performance and energy efficiency variation on Intel processors. In *E2SC'17*.

[35] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. 2014. Evaluation of CPU Frequency Transition Latency. *Comput. Sci.* 29, 3-4 (Aug. 2014), 187–195.

[36] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014).

[37] Microsoft Corp. 2017. Power and performance tuning. https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/hardware/power/power-performance-tuning.

[38] Thiago Ramon Goncalves Montoya. 2017. ZenStates. https://github.com/r4m0n/ZenStates-Linux.

[39] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. 2013. Hierarchical Power Management for Asymmetric Multi-core in Dark Silicon Era. In *Proceedings of the 50th Annual Design Automation Conference*. Article 174, 9 pages.

[40] Patrick Mylund Nielsen. 2012. cpuburn. https://patrickmn.com/projects/cpuburn/

[41] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. The Hipster Approach for Improving Cloud System Efficiency. *ACM Trans. Comput. Syst.* 35, 3, Article 8 (Dec. 2017), 28 pages.

[42] Tapti Palit, Yongming Shen, and Michael Ferdman. 2016. Demystifying Cloud Benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software*. 122–132.

[43] Jacob Pan. 2013. RAPL (Running Average Power Limit) driver.

[44] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. 2008. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. 48–59.

[45] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. 2011. Energy Management in Mobile Devices with the Cinder Operating System. In *Proc. EuroSys*.

[46] Robert Schöne, Daniel Molka, and Michael Werner. 2015. Wake-up Latencies for Processor Idle States on Current x86 Processors. *Comput. Sci.* 30, 2 (May 2015), 219–227.

[47] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. 2013. Power containers: an OS facility for fine-grained power and energy management on multicore servers. In *Proc. 18th ASPLOS*. 65–76.

[48] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. 2009. Koala: A Platform for OS-level Power Management. In *Proceedings of the 4th ACM European Conference on Computer Systems*. 289–302.

[49] E. Del Sozzo, G. C. Durelli, E. M. G. Trainiti, A. Miele, M. D. Santambrogio, and C. Bolchini. 2016. Workload-aware Power Optimization Strategy for Asymmetric Multiprocessors. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. 531–534.

[50] UEFI Forum. 2016. Advanced Configuration and Power Interface Specification, Version 6.1. http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf.

[51] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*. 213–224.

[52] Augusto Vega and Heather Hanson. 2013. Crank It Up or Dial It Down : Coordinated Multiprocessor Frequency and Folding Control. *Proceedings of the 46th Annual International Symposium on Microarchitecture* (2013), 210–221.

[53] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems*.

[54] Carl Waldspurger. 2002. Memory resource management in VMware ESX server. In *Proc. of the 2002 Symp. on Operating Systems Design and Implementation*.

[55] Carl A. Waldspurger and William E. Weihl. 1994. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 1–11.

[56] Q. Wu, Q. Deng, L. Ganesh, C. H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song. 2016. Dynamo: Facebook's Data Center-Wide Power Management System. In *Proceedings of 43rd Annual International Symposium on Computer Architecture*. 469–480.

[57] Rafael J. Wysocki. 2017. intel_pstate CPU Performance Scaling Driver. https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html.

[58] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. 2002. ECOSystem: Managing Energy As a First Class Operating System Resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[59] Huazhe Zhang and H Hoffman. 2015. A quantitative evaluation of the RAPL power control system.