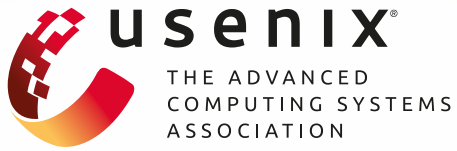

Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale

Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan, *Google LLC*

https://www.usenix.org/conference/osdi20/presentation/li-shaohong

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale

Shaohong Li, *Google LLC*
Xi Wang, *Google LLC*
Xiao Zhang, *Google LLC*
Vasileios Kontorinis, *Google LLC*
Sreekumar Kodakara, *Google LLC*
David Lo, *Google LLC*
Parthasarathy Ranganathan, *Google LLC*

Abstract

As the demand for data center capacity continues to grow, hyperscale providers have used power oversubscription to increase efficiency and reduce costs. Power oversubscription requires power capping systems to smooth out the spikes that risk overloading power equipment by throttling workloads. Modern compute clusters run latency-sensitive serving and throughput-oriented batch workloads on the same servers, provisioning resources to ensure low latency for the former while using the latter to achieve high server utilization. When power capping occurs, it is desirable to maintain low latency for serving tasks and throttle the throughput of batch tasks. To achieve this, we seek a system that can gracefully throttle batch workloads and has task-level quality-of-service (QoS) differentiation.

In this paper we present Thunderbolt, a hardware-agnostic power capping system that ensures safe power oversubscription while minimizing impact on both long-running throughput-oriented tasks and latency-sensitive tasks. It uses a two-threshold, randomized unthrottling/multiplicative decrease control policy to ensure power safety with minimized performance degradation. It leverages the Linux kernel's CPU bandwidth control feature to achieve task-level QoS-aware throttling. It is robust even in the face of power telemetry unavailability. Evaluation results at the node and cluster levels demonstrate the system's responsiveness, effectiveness for reducing power, capability of QoS differentiation, and minimal impact on latency and task health. We have deployed this system at scale, in multiple production clusters. As a result, we enabled power oversubscription gains of 9%–25%, where none was previously possible.

1 Introduction

Data centers form the backbone of popular online services such as search, streaming video, email, social networking, online shopping, and cloud. The growing demand for online services forces hyperscale providers to commit massive capital to continuously expand their data center fleet. The overall capital expenditures for just the top 5 hyperscale providers (Amazon, Google, Microsoft, Facebook, Apple) in 2019 reached $120B out of a total $210B for all data centers worldwide [10, 11]. The majority of these investments are allocated towards buying and building infrastructure, such as buildings, power delivery, and cooling, to host the servers that compose the warehouse-scale computer. Power oversubscription is the practice of deploying more servers in a data center than the data center's power supply can nominally support if all servers were 100% utilized. Power oversubscription allows deploying more servers into a data center, and therefore reduces the number of data centers needed to be built. The cost savings potential of power oversubscription amounts to billions of dollars per year and is therefore of great importance to data center operators.

However, power oversubscription comes with a risk of overload during power peaks, and thus often comes with protective systems such as power capping. Power capping systems enable safe power oversubscription by preventing overload during power emergencies. Power capping actions include suspending low-priority tasks [18], throttling CPU voltage and frequency using techniques such as dynamic voltage and frequency scaling (DVFS) and running average power limit (RAPL) [8, 13, 25], or packing threads in a subset of available cores [17]. The action needs to be compatible with the workloads and meet their service-level objectives (SLOs). This, however, is challenging for clusters with throughput-oriented workloads co-located with latency-sensitive workloads on the same servers.

Throughput-oriented tasks represent an important class of computation workloads. Examples are web indexing, log processing, and machine learning model training. These workloads typically have deadlines on the order of hours for when the computation needs to be completed, making them good candidates for performance throttling when a cluster faces a power emergency due to power oversubscription. Nevertheless, missing the deadline can result in serious consequences such as lost revenue and diminished quality, thus making them unamenable to interruption.

Latency-sensitive workloads are a different class. They need to complete the requested computation on the order of milliseconds to seconds. A typical example is an application that handles user requests. High latencies result in bad user experience, eventually leading to loss of users and revenue. Unlike throughput-oriented pipelines, such tasks are not amenable to performance throttling. They often are considered high priority and need to be exempt from power capping.

In our data centers, throughput-oriented and latency-sensitive tasks are co-located on the same server to increase resource utilization [20]. This introduces a need for a fine-grained power capping mechanism that throttles the performance of throughput-oriented tasks to reduce server power usage while exempting high-priority latency-sensitive tasks.

This paper describes a simple, robust, and hardware-agnostic power capping system, Thunderbolt, to address these challenges. It throttles the CPU shares of throughput-oriented workloads to slow them down "just enough" to keep power under specified budget, while leaving latency-sensitive tasks unaffected. It has been deployed in large-scale production data centers.

To our knowledge, Thunderbolt is the first industry system that simultaneously achieves the following goals. All of these are important to scale out mission critical systems.

- A system architecture that enables oversubscription across large power domains. Power pooling and statistical multiplexing across machines in large power domains maximizes the potential for power oversubscription.
- Quality-of-service-aware, hardware-agnostic power throttling mechanism with wide applicability. Our system relies only on established Linux kernel features, and thus is hardware platform agnostic. This enables the flexibility of introducing a variety of platforms into data centers without compromising the effectiveness of power capping. Our task-level mechanisms allow differentiated quality-of-service (QoS). Specifically, Thunderbolt does not affect serving, latency-sensitive workloads co-located with throughput-oriented workloads on the same server, and has the ability to apply different CPU caps on workloads with different SLOs. The platform-agnostic and QoS-aware nature allows the system to be tailored to the requirements of a wide spectrum of hardware platforms and software applications.
- Power safety with minimized performance degradation. A two-threshold scheme with a randomized unthrottling/multiplicative decrease algorithm enables minimal performance impact while ensuring that a large amount of power can be shed to avoid power overload during emergencies.
- System availability in the face of power telemetry unavailability. Power telemetry availability has not drawn much attention in most previous power capping systems, but we found it to be the availability bottleneck in our system. Thunderbolt introduces a failover subsystem to maintain power safety guarantees even when power telemetry is unavailable.

We have deployed this system in multiple data centers over a period of two years. We have verified proper operation at scale and achieved oversubscription of 9–25% when none was previously possible. At such an oversubscription level, data centers run at high power efficiency and are close to the edge of exceeding their power limits. Power capping is expected to occur a few times a year. Section 7 has more details.

2 Background

Warehouse-sized data centers run very complex and diverse workloads and need a flexible power actuator to handle complicated application scenarios. Google recently published a power capping system [18] that intentionally suspends low-priority tasks which often results in task timeouts and failures. Such interruption is appropriate in certain situations; for instance, some tasks can tolerate occasional downtime but prefer to have consistent performance when they run, and prefer to be interrupted so they can be rescheduled somewhere else rather than being slowed down. However, for throughput-oriented workloads, this is not only wasteful of compute resources but is also disruptive.

Popular software frameworks like Hadoop [19], MillWheel [3], and TensorFlow [1] provide checkpointing functionality to allow tasks to handle failures gracefully. Checkpointing itself, however, incurs non-negligible cost and complexity. Users have to balance between the risk of failure and the overhead of runtime checkpointing. Even with checkpointing, some amount of work is wasted when a task is killed and restarted. For distributed computing that requires synchronization among workers (e.g., synchronized machine learning training), a killed task can easily become a straggler as others have to wait for it to make forward progress. Our system aims to provide a more graceful solution where traditional task killing or suspension is too costly.

Most previous studies control CPU power to affect overall machine power draw. Our system follows this practice, because CPU power draw is much higher than that of memory or storage components (e.g., flash and disk) on the commodity servers in our data centers.

Data center workloads also run at different priorities with varying QoS. It is highly desirable to reflect differentiated QoS even under power capping. Previous industry power capping systems, such as Dynamo [25] and CapMaestro [14], differentiate priorities at the machine level. They assign priorities to individual machines and build a global priority-aware control policy for all machines involved. To provide QoS

differentiation with Dynamo or CapMaestro, tasks with different capping priorities have to be scheduled on different machines. This conflicts with our requirement to run mixed-priority workloads on the same machine to improve resource utilization. In contrast, our approach is designed to provide QoS differentiation when workloads of different priorities run on the same machine.

From a scheduling perspective, our problem may look similar to the classic problem of scheduling latency-sensitive and throughput-oriented tasks on the same machine and optimizing for latency and throughput. However, it is a different form of the problem to which existing scheduling solutions do not directly apply. The constraining resource is power, which neither cluster scheduler nor local node scheduler can directly control or allocate. Instead we control power indirectly by controlling CPU usage. We treat power as a system output, measure it via power meters, and feed it back into the system to build a control loop. We have to care about the availability of power readings that are external signals. Violation of the power budget results in not performance degradation but high-stake physical failures (tripping circuit breakers) and immediate power loss to thousands of machines. Therefore a strong guarantee of power not exceeding the budget is the top priority, requiring fast response and a wide dynamic range of power control. Optimization for latency and throughput must not compromise this guarantee.

3 Terminology

To facilitate the explanation of our system, we define a few key terms summarized here for easy reference.

Thunderbolt. The power capping system as a whole, named after the resulting power curves that look like a thunderbolt (see Figure 5). The overall architecture is described in Section 4.

Load shaping. The "reactive capping" subsystem and closed-loop control policy using power signals for fine-grained power capping control. It is described in Section 4.1.2. It uses CPU bandwidth control (described below) as the node-level mechanism.

CPU bandwidth control. The node-level mechanism for load shaping. It leverages the CPU bandwidth control feature provided by Linux's completely fair scheduler to throttle the CPU usage of tasks. It is described in Section 4.1.1.

CPU jailing. The "proactive capping" backup subsystem that takes over when power signals are unavailable and load shaping cannot function. It includes an open-loop control policy of risk assessment and a node-level mechanism that makes use of Linux's CPU affinity features to limit machines' CPU utilization. It is described in Section 4.2.1.

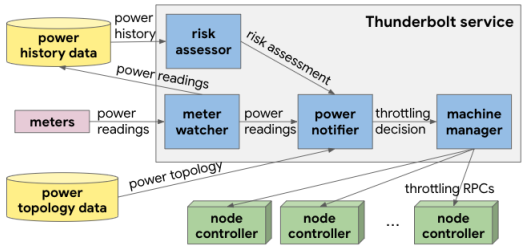

Figure 1: Software architecture of Thunderbolt.

4 Architecture and Implementation

Thunderbolt is capable of performing two types of end-to-end power capping actuation tailored to throughput-oriented workloads: a primary mechanism called reactive capping, and a failover mechanism called proactive capping. Reactive capping monitors real-time power signals read from power meters and reacts to high power measurements by throttling workloads. When power signals become unavailable, e.g., due to meter downtime, proactive capping takes over and assesses the risk of breaker trips. The assessment is based on factors such as power in the recent past and for how long the signals have been unavailable. If the risk is deemed high, it proactively throttles tasks.

The reactive capping system depends on power signals provided by power meters installed close to the protected power equipment, like circuit breakers. Meters are installed at every power "choke point" whose limit will be first reached as power draw increases. In our data centers the choke points are typically power distribution units (PDUs) or medium voltage power planes (MVPPs) [18]. This differs from the more widely adopted approach of collecting power measurements from individual compute nodes and aggregating at upper levels. Our approach has several advantages. It is simple. It avoids aggregation and the associated data quality issues such as time unalignment and partial collection failures. It also avoids the need to estimate power drawn by non-compute equipment, such as data center cooling, that does not provide power measurements.

Figure 1 illustrates the software architecture of Thunderbolt. The meter watcher module polls power readings from meters at a rate of one reading per second. It passes the readings to the power notifier module and also stores a copy in a power history datastore. The power notifier is a central module that implements the control logic of reactive and proactive capping. When power readings are available, it uses the readings for the reactive capping logic. When the readings are unavailable, it queries the risk assessor module for the proactive capping logic. The risk assessor uses the historical power information from the power history datastore to assess the risk of breaker trips. If either logic decides to cap, the power notifier will pass appropriate capping parameters to the ma-

chine manager module, which then sends remote procedure call (RPC) requests to the node controller of individual machines concurrently to reduce power. Important data about the power delivery topology, such as the protected power limits and the machines to be throttled under the power domain, are obtained from a power topology datastore.

The scale of a power domain can vary from a few megawatts, such as a PDU, to tens of megawatts, such as a MVPP. One instance of Thunderbolt is deployed for each protected power domain. The instance is replicated for fault tolerance. There are 4 replicas in a 2-leader, 2-follower configuration. Only the leader replicas can read power meters and issue power shedding RPCs. The node controller's power shedding RPC services are designed to be idempotent and can handle duplicate RPCs from different leader replicas. We require two identical leader replicas to ensure power shedding is available even during leader election periods. Followers take over when leaders become unavailable.

The architecture allows Thunderbolt to scale easily. When a new power domain is turned up, a new Thunderbolt instance can be deployed without affecting existing instances for other domains. When machines are added to or removed from a power domain, only the power topology data needs to be updated to include an up-to-date list of machines.

4.1 Primary subsystem: reactive capping

4.1.1 Node-level mechanism: CPU bandwidth control

CPU usage is a good indicator for the CPU power drawn by a running task. We use the CPU bandwidth control feature of the Linux completely fair scheduler (CFS) [21] to precisely control the CPU usage of tasks running on a node, in order to control the power drawn by the node.

Individual tasks run inside their own Linux control groups (cgroups). The Linux scheduler provides two parameters for a cgroup, namely *quota* and *period*. Quota controls the amount of CPU time the workload gets to run during a period and is replenished every period. Quota is shared and enforced by all logical CPUs in the system. The quota and period can be set for each cgroup and are typically specified at millisecond granularity. A separate (per cgroup and per logical CPU), cumulative *runtime_remaining* variable is kept inside the kernel. The cumulative (per logical CPU) *runtime_remaining* is consumed when a thread is running on the CPU. When it reaches zero, it attempts to draw from the per-cgroup quota pool. When the quota pool is empty, the running thread is descheduled and no thread in the same cgroup can run until quota is replenished at the beginning of the next period.

We track the historical CPU usage of all workloads running on the machine. During a capping event, every node in the power domain will receive an RPC to throttle throughput-oriented workloads. The RPC contains parameters describing how much the CPU usage of the tasks should be reduced (details in Section 4.1.2). The node controller that receives the RPC uses the historical CPU usage of all throughput oriented workloads to determine how much CPU time to throttle. The new quota and period values are then calculated and configured for each cgroup on the machine.

Different tasks have different cgroups, and we can achieve task-level QoS differentiation by adjusting their cgroup parameters. The Thunderbolt framework is capable of assigning different CPU throttling levels to different cgroups, with more restrictive levels to lower priority cgroups. In our cluster resource management systems, throughput-oriented tasks are typically assigned low priorities while latency-sensitive tasks are assigned high priorities. CPU throttling is applied only to cgroups of throughput-oriented tasks, exempting cgroups of latency-sensitive tasks. This is appropriate for our production environment, where a significant portion of total CPU resources is consumed by throughput-oriented tasks (also known as batch tasks [20]), and it is undesirable to throttle business-critical latency-sensitive tasks. Exempting all latency-sensitive tasks comes with a caution about non-sheddable power, which is explained below. Kernel threads are also exempt from throttling and their CPU usage is very low compared to regular tasks.

Non-sheddable power, the lower bound of the power control range, is an important consideration for power oversubscription and capping. With our implementation of Thunderbolt, non-sheddable power can be attributed to CPU usage by exempt tasks, machine idle power, and other uncontrolled power users such as cooling equipment. We deliberately set the oversubscription level so that non-sheddable power does not exceed the protected power limits. We run continuous monitoring and rigorous analysis to predict and alert on the portion of non-sheddable power in our data centers. In an unlikely event when high non-sheddable power is predicted in a cluster, site operators can leverage global load balancing to redirect traffic of latency-sensitive tasks elsewhere to offset the risk.

The relationship between throttling levels and power draw is nonlinear and workload dependent, therefore we always use CPU bandwidth control in conjunction with power metering and negative feedback to ensure expected power reduction is achieved. The feedback loop is described in detail in Section 4.1.2.

Characteristics of CPU bandwidth control. CPU bandwidth control has two important properties:

Platform-agnostic. CPU bandwidth control is a pure software feature supported by the upstream Linux kernel. It can be switched on for almost any new platform with minimal additional effort.

Task-level control. CPU bandwidth control is at the task (cgroup) level. Specifically, tasks of varying priorities are co-located on the same server and can even run on the same physical core. CPU bandwidth control has the

Table 1: Comparison between CPU bandwidth control, DVFS, and RAPL for power limiting.

	CPU bandwidth control	DVFS	RAPL
Response time	1 ms	100 μs	100 μs
Spatial granularity	cgroup	Physical core	Processor socket
Power feedback control	Requires external	Requires external	Processor built-in
Mechanism	Pure software	Requires hardware support	Requires hardware support

required fine-grained visibility and control to provide the differentiated QoS.

These properties make CPU bandwidth control a good fit for our needs. We have also considered other popular hardware-based alternatives, in particular dynamic voltage and frequency scaling (DVFS) and Intel's running average power limit (RAPL). Below we compare and discuss CPU bandwidth control and the two alternatives, and explain why, despite the merits of the two alternatives, we do not adopt them for Thunderbolt.

We summarize several attributes of CPU bandwidth control, DVFS, and RAPL in Table 1. Given their nature of hardware control, DVFS and RAPL both have faster power response times than bandwidth control. In practice, however, we find that the longer response time of CPU bandwidth control is still fast enough to be an effective load shedding mechanism for safe power oversubscription (see Section 6).

CPU bandwidth control vs RAPL: RAPL is available only on Intel platforms. More importantly, the power limit can only be set on a per socket basis, which means it does not provide task-level control granularity. Alternative approaches are possible to achieve differentiated task QoS using RAPL if additional support is added to the node controller. For instance, tasks with different QoS may be scheduled on different sockets. Apart from the extra complexity, such a scheduling constraint has a disadvantage of limiting CPU resource overcommitment opportunity, which is undesirable for our cluster scheduler [22].

CPU bandwidth control vs DVFS: DVFS is available on most modern high-performance platforms, bringing its compatibility close to CPU bandwidth control. However, it may also have problems supporting task-level control. For example, per-core DVFS is supported by Intel only for Haswell and later generations, and it is not supported by some non-x86 vendors. In terms of power control and performance impact, as we will show in Section 5, DVFS is incapable of throttling down to very low power levels but it has better power efficiency than bandwidth control.

Operational factors. The platform-agnostic nature of CPU bandwidth control is vital to new platform introductions. Even if a new microarchitecture supports fine-grained DVFS, driver support for new platforms often have issues that require extra work. More importantly, per-task DVFS setting is not supported by the upstream Linux kernel. It is also not rare to find chip errata that require workarounds. Using CPU bandwidth control as either the main throttling mechanism or as a fallback mechanism removes these uncertainties in the critical path. It makes us more comfortable about scaling up our data centers with heterogeneous processor microarchitectures.

Overall we consider CPU bandwidth control essential to the success of Thunderbolt. In the future DVFS can be added as a node-level optimization. When Thunderbolt was first deployed, per-task DVFS setting was not available in our Linux kernel. We have recently added per-task DVFS support to the Linux kernel to enable additional trade-offs between performance and efficiency on Intel servers. The same kernel mechanism can be used for power throttling.

4.1.2 Control policy: load shaping

The load shaping control policy determines when and how much the actuator (CPU bandwidth control) should throttle CPU usage in order to control power.

Formally, the power draw of a power domain can be written as

$$p(t) = \sum_{i=1}^{N} f_i(c_i(t) + u_i(t)) + n(t) \quad (1)$$

where t is (discrete) time, p is the total power draw, N is the number of machines, f_i is the power drawn by machine i as a monotonic function of the normalized machine CPU utilization (in the range of [0, 1]), c_i is the CPU used by controllable tasks, u_i is the uncontrollable CPU used by exempt tasks and the Linux kernel, and n is the power drawn by non-machine equipment. Our goal is to cap c_i so that $p < l$ for a power limit l. Preventing overload ($p > l$) is the top priority, while keeping p close to l when $p < l$ is also desirable for efficiency.

We use a *randomized unthrottling/multiplicative decrease* (RUMD) algorithm. If $p(t) > l$, then we apply a cap for the CPU usage of each controllable task. The cap is equal to the task's previous usage multiplied by a *multiplier*, m, in the

range of (0, 1). Then the power draw at the next time step is

$$\begin{aligned} p(t+1) &= \sum_{i=1}^{N} f_i(c_i(t+1)+u_i(t+1))+n(t+1) \\ &\leq \sum_{i=1}^{N} f_i(mc_i(t)+u_i(t+1))+n(t+1) \end{aligned} \tag{2}$$

This cap is updated every second, and c_i decreases exponentially with time, until $p < l$. Note that, because of the u_i and n terms, there is no guarantee that $p(t+1) < p(t)$. Nevertheless, as explained in Section 4.1.1, in practice we ensure with high confidence that non-sheddable power is less than the power limit, that is,

$$\sum_{i=1}^{N} f_i(u_i(t))+n(t) < l, \forall t \tag{3}$$

Therefore power will eventually be reduced below the limit.

The system works on a time scale of seconds: power measurements are read once per second, and throttling parameters are updated every second. This is because the typical end-to-end response time is 1–2 seconds from a high power draw to power being sufficiently reduced by throttling. This is mostly attributed to metering delays. We conservatively budget 5 seconds to account for occasionally longer metering delays and network tail latency.

Throttling stops when p decreases to be below l. To avoid fast power surges, throttling should stop in a progressive manner. We do this by removing the CPU cap on a random portion of machines every second. For instance, if it is configured to completely unthrottle all machines in 5 seconds, then a random non-overlapping set of 20% of machines will be unthrottled every second. Alternatively, one may progressively lift the cap in an additive manner for each machine at the same time, leading to an *additive increase/multiplicative decrease* (AIMD) algorithm [5]. We choose a randomized unthrottling scheme instead of AIMD because it is simpler (no need for an additive increase parameter), and AIMD's "fairness" property (machines converging to having the same CPU utilization) is not required for our system, as long as randomization avoids any machine from being disproportionally impacted.

Similar to AIMD, our RUMD algorithm also has the desirable partial distributedness property. The central policy controller requires no detailed system states, such as the CPU usage and task distribution of each machine, other than the total power. The distributed node controllers can make independent decisions based solely on a few parameters that the policy controller sends to all node controllers.

The result of the RUMD algorithm is a power curve oscillating around the capping limit in a sawtooth-like pattern, as can be seen in Section 6.2.

Implementation details. Here we give some details about our implementation of the RUMD algorithm. In particular, we explain how we balance two competing properties, responsiveness for power safety and efficiency for minimizing performance impact, by maintaining two capping thresholds, one high and one low. The *high threshold*, placed close to the protected power limit, is associated with a *hard multiplier* close to 0 in order to quickly reduce power for safety. The *low threshold*, placed with a larger margin from the protected limit, is associated with a *soft multiplier* for gentle throttling.

We start by explaining the high threshold for power safety. Our end-to-end response time budget is 5 seconds. In 5 seconds, we have observed that power in a nearly full cluster will increase by no more than 2% of the protected equipment limit. Therefore we place the high threshold at 98% of the limit. The hard multiplier associated with this threshold is set to be close to 0 for a quick reduction of a large amount of power. This is because the only strong power guarantee is non-sheddable power being less than the limit (Equation 3), and thus sheddable power has to be reduced to nearly zero quickly to guarantee responsiveness and safety.

It is worth noting that most circuit breakers do not immediately trip when their rated power limit is reached. They may tolerate a few seconds to tens of minutes of power overload [9]. In theory we may make use of this time buffer and set the capping threshold at the power limit. However, how long a breaker can sustain a power overload depends on many factors, such as the design of the breaker, the magnitude of the overload, and ambient temperature [9], and is thus hard to predict. Power overload also decreases the equipment's lifetime. Therefore we choose to place the high threshold 2% below the power limit to avoid tapping into the overload region.

We do not reduce the CPU cap of a task below a minimum value (0.01) because the quota value in CPU bandwidth control has to be greater than zero. This has a production implication: when continuous throttling is applied long enough, affected tasks will eventually converge to the minimum CPU share. In this case, while all affected tasks cannot make meaningful progress and power will be low, some tasks can still respond to health checks and survive. Because of this, task failures due to continuous throttling are expected to be fewer than failures caused by completely suspending tasks, as can be seen in Section 6.2.

The hard multiplier close to zero, while being responsive and safe, is not efficient for utilizing the power budget because it leads to power oscillation with a large amplitude. Therefore we introduce the low threshold associated with the soft multiplier. The soft multiplier is close to 1 to improve efficiency at the cost of responsiveness, and the low threshold is placed below the high threshold to allow the longer response time.

We further optimize our design by not activating the low threshold until throttling is triggered, and deactivating it after throttling has not been active for a while. This way power is allowed to reach the range between the two thresholds without throttling.

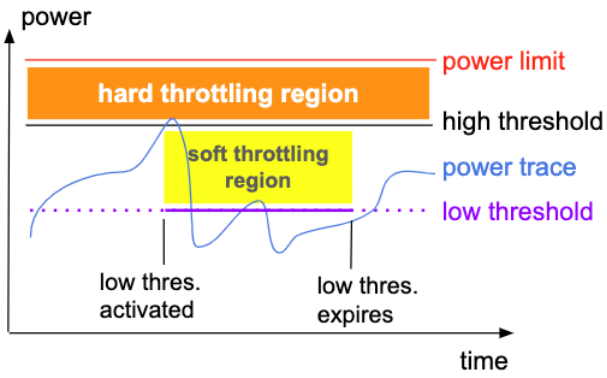

Figure 2: Load shaping power control.

Table 2: Load shaping parameters used in production.

Load shaping parameter	Value
High threshold	98% of protected limit
Low threshold	96% of protected limit
Hard multiplier	0.01
Soft multiplier	0.75
Low threshold expiration	5 minutes
Throttling timeout	1–20 seconds

Randomized unthrottling is implemented by assigning to each machine a random throttling timeout in a range. A random timeout is included in the throttling RPCs and sent to each machine every second to refresh its timeout. When power is below the capping threshold, the machines will stop receiving the RPCs and will unthrottle after the last received timeout has passed. We choose a repeatedly-refreshed timeout instead of a stop-throttling RPC because stop-throttling RPCs may be dropped or even never reach some machines if the network becomes partitioned.

Figure 2 illustrates the power trace in a typical throttling scenario. Table 2 lists the parameters we use in production.

4.2 Failover subsystem: proactive capping

The feedback control of reactive capping relies on power meters. However, power meters and the facility network connecting the meters to the production network are not always available. On average, individual meters and facility network have about 99.9% availability in our data centers, and it varies by location. Transient network issues can cause seconds to minutes of power signal interruption, while meter downtime can be days before the meter gets repaired.

Without power signals, it is not straightforward to use CPU bandwidth control for an open-loop control. We have built models to map machine power utilization to CPU utilization, so we may distribute the power domain's total power budget to individual machines and translate a machine's power budget to a CPU budget. However, it would require a sophisticated algorithm to allocate the machine's CPU budget among individual tasks while respecting the tasks' QoS difference. Instead of introducing a complex algorithm, we implement a simple mechanism, CPU jailing, that specifies a total CPU budget for a machine and leverages the Linux CFS scheduler to provide task QoS differentiation (although the differentiation is relaxed compared to when meter is available, which is explained further below). In a nutshell, CPU jailing is coarser-grained than CPU bandwidth control, but much easier to reason about when power signals are unavailable.

DVFS or RAPL, where supported, may also be used for proactive capping because we only need machine-level control. However, we favor the platform-independent CPU jailing for the same reasons as we favor the platform independence of CPU bandwidth control.

We have also considered collecting power signals from secondary sources, such as the machines' power supply units, or from power models. However, we found that the data quality of the sensors and the accuracy of the models for some hardware do not meet our production requirements.

4.2.1 Node-level mechanism: CPU jailing

CPU jailing masks out ("jails") a certain number of logical CPUs from tasks' runnable CPU affinity [15] to cap total machine power. We refer to the portion of jailed CPUs as *jailing fraction*, denoted by J. CFS will maintain proportional fairness among tasks on the remaining available logical CPUs. Each jailing request comes with a timeout that can be renewed. Once jailing expires, previously masked CPUs will immediately become available to all tasks.

CPU jailing immediately caps peak power draw as it effectively limits maximum CPU utilization to $(1-J)$ on every single machine. It sets an upper bound for power draw, allowing safe operation for an extended time without power signals. Because of increased idleness, jailed CPUs have a higher chance of entering deep sleep to further reduce machine power.

The jailing fraction is uniformly applied to individual machines, regardless of their CPU utilization. Consequently, machines with low utilization are less impacted than highly utilized machines. As an extreme example, CPU jailing might not affect tasks at all on machines with utilization well below $(1-J)$.

Certain privileged processes, such as critical system daemons, are explicitly exempt (i.e., they can still run on jailed CPUs). The rationale is that their CPU usage is very low compared to regular tasks but the consequences of them being CPU starved can be devastating (e.g., machine cannot function correctly). A side effect of exemption is that it puts some sporadic usage on the jailed CPUs and occasionally prevents

them from entering deep sleep state.

A main disadvantage of CPU jailing is the relaxed QoS differentiation. For example, the latency of a serving task can be severely affected when too many cores are jailed. Although this effect is attenuated by the fact that latency-sensitive tasks run at higher priorities and can preempt lower-priority throughput-oriented tasks during CPU resource contention, CPU jailing is less favorable than CPU bandwidth control and is only employed where load shaping is not applicable.

Technically, one may achieve strict QoS differentiation by applying CPU jailing to only throughput-oriented tasks while exempting latency-sensitive ones. However, doing so without power signals is intangible in practice. If latency-sensitive tasks are exempt from CPU jailing, the only strong guarantee we have about power is that non-sheddable power does not exceed power limit (Equation 3). In this situation, guaranteeing power safety would require not running throughput-oriented tasks at all, which we cannot afford.

Determining jailing fraction J. A proper jailing fraction J can be determined from two factors: the relation between CPU utilization and power utilization, and power oversubscription ratio.

For power safety, we need to ensure power is reduced to a safe level after a certain fraction of CPUs are jailed. This value of J can be calculated from the power oversubscription ratio and the CPU utilization-power utilization relation of the given collection of hardware in the power domain, as follows:

$$J = 1 - U_{cpu} = 1 - g_{power \to cpu}(\frac{1}{1+r}) \tag{4}$$

In the formula, U_{cpu} is the highest allowed CPU utilization (normalized to the total CPU capacity), $g_{power \to cpu}$ is a function to convert power utilization (normalized to the theoretical total peak power) to CPU utilization, and r is the oversubscription ratio defined by the extra oversubscribed power capacity as a fraction of the nominal capacity. $1/(1+r)$ gives the maximum safe power utilization, which can be converted to U_{cpu} given that the CPU utilization-power utilization relation is monotonic. A greater r leads to smaller allowed power utilization and smaller U_{cpu}, which in turn leads to greater J.

In production, we set J to 20%–50% depending on a cluster's workloads and risk profiles. This is a deliberate trade-off between performance SLO and power oversubscription opportunity.

4.2.2 Control policy: risk assessment of power signal unavailability

As a fallback approach, CPU jailing is triggered when we lose power measurements from the meters and the risk of power overloading is high. The risk is determined by two factors, predicted power draw and meter unavailability duration. Higher predicted power draw and longer meter unavailability means higher risk. The end-to-end delay from risk assessment to power reduction is typically 1–2 seconds, similar to load shaping. In our implementation, we use a simple and conservative probabilistic model to estimate the probability of power reaching the protected equipment limit during certain meter downtime given the power draw of the recent past. CPU jailing is triggered if the probability is high due to high recent power draw and long enough downtime. Our conservative model favors low false negatives (i.e., CPU jailing is triggered when overload would have happened without it) at the cost of relatively high false positives (i.e., CPU jailing is triggered even when it does not have to). This is appropriate because power safety is our top priority and power reading unavailability is infrequent. The probabilistic model is not the focus of this paper, but one can freely use any model that estimates the risk from any available data and plug it in here.

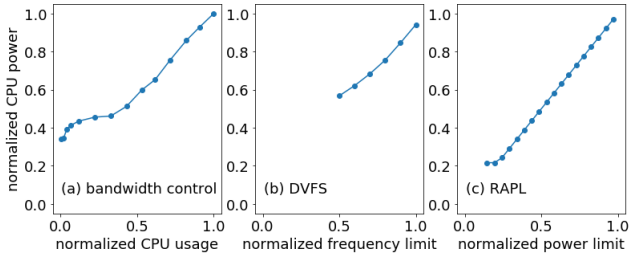

Figure 3: CPU power response to bandwidth control, DVFS, and RAPL.

5 Evaluation Results at the Node Level

Before discussing data center-level aggregated data, we show two examples of node-level data from experiments performed on an Intel Skylake CPU.

CPU power and set point. To quantify the effectiveness of CPU bandwidth control, DVFS, and RAPL to control power, we measure total CPU power under various set points of the three knobs. We ran Intel's "power virus" workload [7] that stresses the CPU and the memory to maximize power draw. We then separately used CPU bandwidth control, DVFS, and RAPL to limit CPU power and compared the results, which are shown in Figure 3. CPU power is normalized to the highest power observed when none of the power management mechanisms are enabled.

Figure 3(a) shows that, with CPU bandwidth control, we are able to reduce CPU power to 0.34 due to significant deep sleep state residency from bandwidth control.

In comparison, Figure 3(b) shows that with DVFS, power draw is still relatively high at 0.57 when the lowest frequency limit is applied. The frequency limit is normalized to the base frequency of the processor.

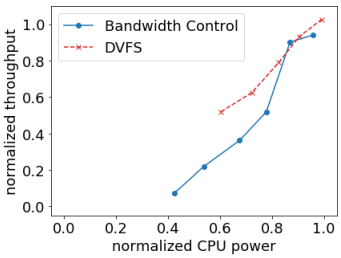

Figure 4: Throughput of a video transcoding task as a function of CPU power under bandwidth control and DVFS.

Figure 3(c) shows RAPL has the widest power reduction range among the three. It is able to reduce power to 0.22. However, we noticed system management tasks were sluggish to respond when RAPL approached the lowest power limits, which suggests higher machine timeout risks if these limits were actually used in practice. By contrast, CPU bandwidth control used in our system only throttles throughput-oriented tasks and the system management tasks are not affected. Thanks to its built-in feedback loop, RAPL is fairly accurate in achieving the provided power budget [26]. RAPL's predictability is an advantage over DVFS or CPU bandwidth control.

CPU power and throughput. In this experiment, we run a throughput-oriented video transcoding task under various set points of CPU bandwidth control and DVFS, and measure CPU power and task throughput. This gives us information about the throughput impact of the two mechanisms under a power budget. Throughput is calculated as the reciprocal of the wall clock time of completing the task, normalized to the throughput where no power throttling is applied. CPU power is normalized to the highest power observed when power virus is run and no power throttling is applied (matching Figure 3).

Results are shown in Figure 4. Throughput is only mildly affected when power is greater than 0.85 for both bandwidth control and DVFS. Possibly memory bandwidth, rather than CPU bandwidth, is the bottleneck in this region. Throughput drops notably as power drops below 0.85 for both mechanisms, but DVFS has higher throughput than bandwidth control under the same power. Therefore, DVFS is more power efficient than bandwidth control. However, in terms of power control dynamic range, DVFS can only reduce power by 40% when the lowest frequency limit is applied, whereas bandwidth control is capable of nearly 60% power reduction. This is consistent with the power virus result in Figure 3. Load shaping events happen infrequently in our data centers, thus power efficiency is not a major factor for our use case.

6 Evaluation Results at Data Center Scale

To characterize the system at scale, we performed experiments in clusters comprising tens of thousands of machines running

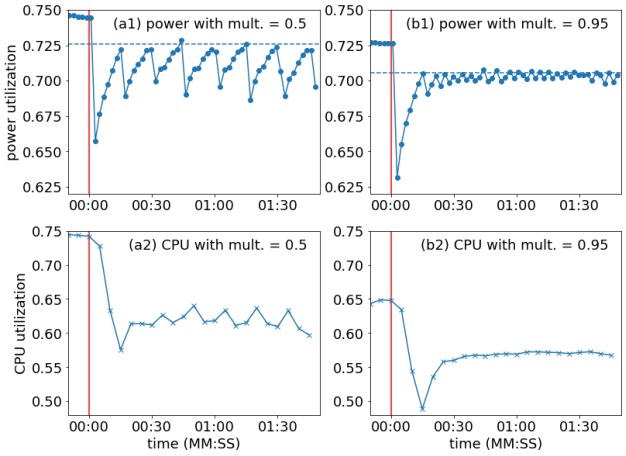

Figure 5: Typical load shaping patterns. (a1) and (a2) show the normalized power and CPU utilization of a load-shaped power domain, with 0.01 hard multiplier and 0.5 soft multiplier. (b1) and (b2) show similar data for the same power domain but with 0.01 hard multiplier and 0.95 soft multiplier. The blue horizontal dashed lines are low power thresholds associated with the soft multipliers. The red vertical lines mark the start of load shaping. (The power and CPU readings are not exactly time-aligned due to sampling delays.)

diverse production workloads in our data centers. Throttling was manually triggered with various combinations of parameters. Power data is collected from data center power meters, which is the same data that Thunderbolt also uses. Power measurement data is normalized to the power domain's equipment limit.

Other metrics are sampled from individual machines and aggregated at the same power domain level corresponding to the power readings. Machine metrics such as CPU usage are normalized to the total capacity of all machines in the power domain unless specified otherwise. Task failures are normalized to the total number of affected tasks.

Latency data are collected from low-level storage services that read and write Linux files and support Google's distributed file system. They are critical services widely deployed in our data centers, running at high priorities and thus exempt from load shaping. They are not exempt from CPU jailing but have high priority to access the remaining CPUs.

6.1 Load shaping in typical scenarios

In this experiment, we picked a production cluster that is dominated by throughput-oriented workloads to test the typical behavior of load shaping. Load shaping was triggered by manually lowering the high power threshold to be just below the ongoing power draw of a power domain.

Power and CPU usage patterns. Figure 5(a1) shows a typi-

Table 3: Load shaping duration, task failure fraction, and 99%-ile read latency of storage services under different scenarios.

	Duration	Failure fraction	Latency
Baseline	25 min.	0.00002	79 ms
0.95 soft mult.	5 min.	0.00000	79 ms
0.75 soft mult.	10 min.	0.00003	80 ms
0.5 soft mult.	5 min.	0.00007	78 ms

cal load shaping pattern of power oscillating around the low threshold. Seconds after throttling is triggered, power is reduced by a large margin because of the hard multiplier. Meanwhile the low threshold is activated. Throttling is gradually lifted as power drops below the low threshold, and power goes back up until it reaches the low threshold. Then power is reduced again, but by a smaller margin because of the soft multiplier. The process continues as throttling is turned on and off repeatedly, resulting in power oscillating around the low threshold. Figure 5(b1), as compared to (a1), shows a soft multiplier closer to 1.0 leads to oscillations of a smaller amplitude, as expected. The response time from load shaping triggering to significant power reduction is about 2 seconds.

Figure 5(a2) and (b2) show the CPU utilization corresponding to (a1) and (b1) respectively. At the shown CPU utilization level, about 0.1 reduction of CPU utilization is needed to reduce 0.02 of power.

Task failures. While tasks are slowed down, we want to ensure that most of them do not fail because of CPU starvation or unexpected side effects. Table 3 shows task failure fractions (the number of failed tasks normalized to the total number of affected tasks) of the same power domain under load shaping with various soft multipliers. "Baseline" indicates no throttling and serves as the baseline for comparison. All load shaping events have a hard multiplier of 0.01 (not shown in the table) while the soft multiplier varies from 0.5 to 0.95. Clearly load shaping does not cause noticeably more failures. The failure fraction remains low compared to the baseline.

Latencies. To assess load shaping's effect on the latencies of latency-sensitive tasks, we inspect the tail 99%-ile read latency of latency-sensitive storage services, shown in Table 3. As expected, the latency is not affected by load shaping because the tasks are exempt from the mechanism.

Differentiation of QoS. To test Thunderbolt's ability to differentiate QoS, we classified tasks into two groups based on their priority, and load-shaped the low-priority group while exempting the high-priority group. Figure 6 shows the total power draw and CPU usage of the two groups of tasks, during the event. The CPU usage of the shaped and the exempt group is reduced by about 0.1 and 0.03, respectively. The exempt group is indirectly affected because the tasks in the

Figure 6: Power and CPU utilization during a load shaping event with multiplier 0.1 that directly affects a subset of tasks. The red vertical line marks the start of the event. The CPU reduction of the load-shaped tasks are more prominent than that of the exempt tasks. The exempt tasks are indirectly affected because of their interaction with the shaped tasks. (The power and CPU readings are not exactly time-aligned due to sampling delays.)

two groups are production tasks with complex interactions. One of such interactions is that a high-priority controller task in the exempt group coordinates low-priority workers in the shaped group, and the controller task has less work to do and consumes less CPU when the workers are throttled. Nevertheless, the ability of load shaping to differentiate tasks is evident.

6.2 Load shaping pushed to the limit

In typical scenarios, as demonstrated in Section 6.1, load shaping reduces power to a safe level just below the threshold and allows power to oscillate around it. However, in extreme cases where power stays above the threshold, the system will need to continuously reduce tasks' CPU usage, eventually to the preset minimum value. The affected tasks will essentially be stopped and make no forward progress. For example, power may remain high after throttling is triggered because new compute-intense tasks are continuously scheduled, or many high-priority tasks exempt from the mechanism spike in their CPU usage. In such cases it is the right trade-off to stop the low-priority tasks in order to prevent power overloading.

To test the behavior of load shaping in such extreme scenarios, we picked a cluster with some low-priority, non-production, throughput-oriented workloads and applied a multiplier continuously to those tasks. (Most of the tasks in that cluster are high-priority, which we exempt from this test thanks to load shaping's ability to differentiate tasks.) We com-

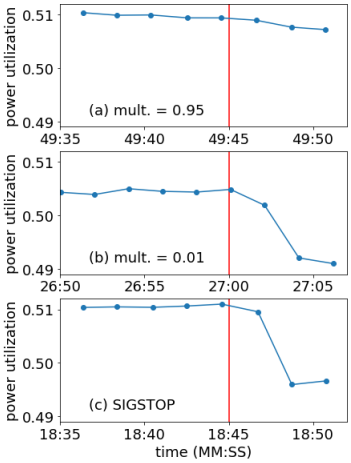

Figure 7: Power responsiveness of continuous throttling and of SIGSTOP. (a) and (b) are throttling with a multiplier of 0.95 and 0.01, respectively. (c) is SIGSTOP. The red vertical lines mark the start of throttling and SIGSTOP.

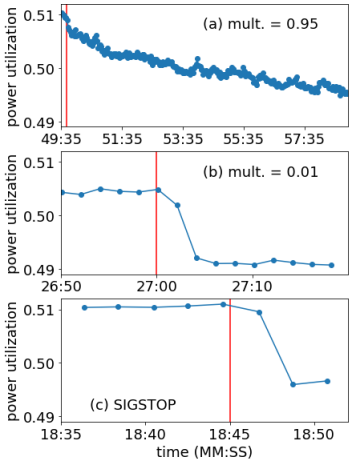

Figure 8: Power reduction by continuous throttling and by SIGSTOP. (a) and (b) are throttling with a multiplier of 0.95 and 0.01, respectively. (c) is SIGSTOP. The red vertical lines mark the start of throttling and SIGSTOP. The x axes of the sub-figures are scaled differently because the power reduction happens at different time scales.

pared continuous throttling to explicitly stopping the tasks by sending them a SIGSTOP signal followed by a SIGCONT signal after 60–75 seconds.

Power responsiveness and range of control. Figure 7 shows the power responsiveness of continuous throttling and of SIGSTOP. Here power is reduced noticeably in 2 seconds. This is true for all the tested multipliers as well as for SIGSTOP. In 4 seconds, about 3% of power is shed by throttling with a 0.01 multiplier and by SIGSTOP, and 0.5% by throttling with a 0.95 multiplier, respectively.

If throttling is applied continuously, we expect tasks to eventually have close-to-zero CPU shares and we achieve similar power reduction as SIGSTOP. This is indeed true. Figure 8 compares the power reduction by continuous throttling with two multipliers, and compares them to SIGSTOP. It plots the same data as Figure 7 but on a larger time scale to show the power reduction. (Note that the x axes of the sub-figures are scaled differently because the power reduction happens at different time scales.) Power is reduced at a slower pace with a multiplier closer to 1, but given enough time it is eventually reduced by an amount similar to SIGSTOP (about 0.015) regardless of multiplier. This is expected, because the cumulative effect of applying any multiplier between 0 and 1 should eventually converge to CPU shares that are close to zero. This also implies that the selection of the multiplier does not affect the effectiveness of power reduction in terms of sheddable power. The selection of the multiplier does affect responsiveness, however, which is important when power spikes need to be throttled quickly.

Task failures. Table 4 lists the task failure fractions during the test periods of continuous throttling and SIGSTOP. "Baseline" indicates no throttling or SIGSTOP and serves as baseline for comparison. Throttling with a 0.95 multiplier has mild effect on failure fraction and can be continuously applied to tasks for longer time (20 minutes here). Both throttling with a 0.01 multiplier and SIGSTOP were only performed for a short period of time (2 minutes), but they caused skyrocketed failure fraction by one to two orders of magnitude. The failures are mostly attributed to tasks being terminated because they fail to respond to health checks. The increased failure fraction of continuous throttling with a 0.01 multiplier is contrasted with the low failure fractions of load shaping in Table 3. Those load shaping events in Table 3 had a 0.01 hard multiplier in effect only for a few seconds, because the hard multiplier was progressively lifted in seconds after power drops below the high power threshold. The failure fraction of continuous throttling with a 0.01 multiplier is one order of magnitude lower than that of SIGSTOP because the throttled tasks still have a minimum CPU share, and some of them can respond

Table 4: Power shedding duration, task failure fraction, and 99%-ile read latency of storage services under different scenarios.

	Duration	Failure fraction	Latency
Baseline	15 min.	0.0007	126 ms
0.95 mult.	20 min.	0.0007	122 ms
0.01 mult.	2 min.	0.003	125 ms
SIGSTOP	2 min.	0.06	135 ms

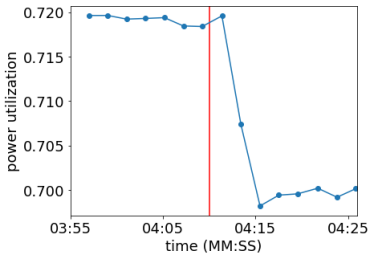

Figure 9: Responsiveness and power reduction of CPU jailing with 20% jailing fraction. Power is reduced by 0.02 in 5 seconds when the power domain's CPU utilization is about 60% (not shown in the figure).

Table 5: Effect of 20% CPU jailing on machine CPU utilization.

	Duration	Machine CPU utilization		
		50%ile	95%ile	99%ile
Baseline	60 min.	0.58	0.80	0.94
CPU jailing	55 min.	0.55	0.69	0.75

to health checks and survive.

Latencies. Table 4 shows the tail 99%-ile read latency of latency-sensitive storage services. As expected, the latency is not notably affected by either load shaping or SIGSTOP, both of which are not applied to those services.

6.3 CPU jailing

For this experiment of CPU jailing, we picked the same production cluster as in Section 6.1, which is dominated by throughput-oriented workloads. We manually performed CPU jailing with a 0.2 jailing fraction, denoted by "20% CPU jailing", and collected data for power, CPU usage, CPU cores in deep sleep states, task failures, and latencies. The same types of data were collected during a period before the CPU jailing event; those data will serve as the baseline for comparison.

Power responsiveness. For the purpose of a failover mechanism, response time is not a concern in most cases, except for the corner case where meter signals are lost while load shaping is, or very close to being, active. Nevertheless, Figure 9 shows that power utilization is reduced by 0.02 (from 0.72 to 0.70) in 5 seconds under 20% CPU jailing. The power reduction is relatively small, because most machines had lower than 80% CPU utilization even before 20% jailing was applied to them. This can be seen in Table 5, discussed further below.

CPU usage. CPU jailing affects machines with high CPU utilization more than those with low CPU utilization. This is evident from Table 5. The median machine CPU utilization without CPU jailing is 0.58, and it is only mildly affected by 20% CPU jailing that limits available machine CPU capacity to 80%. In contrast, the 99%-ile and 95%-ile machine CPU utilizations, which are close to or higher than 80%, are reduced significantly during CPU jailing.

While CPU jailing is a pure software mechanism, it can get extra benefits with hardware support that puts idle cores in power-saving states. In our experiment with 20% CPU jailing, 5% of affected CPU cores entered deep sleep states (C6/C7 states) as compared to 1% of cores without jailing. Noticeably, although 20% of cores are jailed, the portion of deep-sleep cores is always less than 20% due to processes exempt or unaffected by the mechanism.

Table 6: Task failure fraction and 99%-ile read latency of storage services under 20% CPU jailing.

	Duration	Failure fraction	Latency
Baseline	60 min.	0.00003	79 ms
CPU jailing	55 min.	0.00002	86 ms

Task failures and latencies. Table 6 shows the task failure fraction and the 99%-ile read latency of storage services of a power domain in a 20% CPU jailing event. There is no notable difference in failure fraction and latency compared to the baseline. Both latencies are far below our SLO. However, in a separate experiment of 80% CPU jailing we observed an order of magnitude higher latency (not shown in the table), which is not surprising because severe CPU contention is expected with such heavy jailing.

7 Deployment at Scale and Benefits

Thunderbolt has been deployed at scale in our logs processing clusters and has enabled 9%–25% power oversubscription relative to the nominal capacity, depending on the power delivery architecture. The oversubscription is determined by an SLO with the clusters' stakeholders about the expected occurrence frequency of throttling events under realistic worst conditions. Other throughput-oriented clusters, such as web indexing, are also in scope of more aggressive power oversubscription with Thunderbolt.

Logs processing workloads are mostly throughput-oriented and continuously running. Resources are provisioned to accommodate worst-case daily throughput demands, and any disruptive power capping actuation on the worst day is a waste of the resources and cancels the benefits of aggressive power oversubscription. Thunderbolt, by gently throttling computation, distributes the actual work throughout the day, gracefully allowing throughput to be conserved. Despite that most workloads are throughput-oriented, there are still critical latency-sensitive workloads such as low-level storage services, and

therefore QoS differentiation is important.

The reactive capping mechanism has been activated three times by exceptionally high power draw in three production clusters in the first 130 days of year 2020. The proactive capping mechanism has been activated two times by power telemetry unavailability in two high-power production clusters in the same period of time. Such incidents could have resulted in tripping data center breakers without the protection from the power capping system. The activation events went unnoticed by stakeholders, with negligible adverse effect on production.

8 Challenges and Future Work

Thunderbolt, implemented as described in this paper, is suitable for our production clusters running a mix of throughput-oriented and latency-sensitive workloads. Those clusters have a sizable portion of power drawn by throughput-oriented tasks, and a stable usage pattern of latency-sensitive workloads. Therefore, we are able to set an appropriate oversubscription level with high confidence that non-sheddable power will not pose a risk, and that CPU jailing will not starve latency-sensitive tasks. Nevertheless, the Thunderbolt framework is flexible enough for extension and optimization to accommodate clusters of different workload patterns. Here we discuss some directions and challenges.

Our implementation exempts all high-priority latency-sensitive workloads from load shaping but this is not always required. In clusters where latency-sensitive workloads may use too much power, one could further break them down into multiple priority buckets and throttle them as appropriate under their SLOs. Doing so in practice is a challenge as latency-sensitive tasks are generally not amenable to CPU throttling. It will likely require a co-design of throttling policy, SLO, and software infrastructure. For example, one could have an SLO that permits affecting the latencies for a small fraction of time, and design the workload and software infrastructure to respond to high latency properly. For cloud data centers where the infrastructure owner has limited control over the workloads, cloud providers may carefully design service-level agreements (SLAs) to allow throttling "abusive" behaviors, and possibly use price incentives to encourage "good" behaviors.

Thunderbolt sheds power by controlling CPU usage. This may not be effective if the majority of power is used by non-CPU components, such as hardware accelerators. While hardware support is needed to effectively throttle such components, the Thunderbolt software architecture and control policies of load shaping and proactive capping can be adapted to control additional hardware throttling knobs. QoS differentiation will depend on the control granularity of the hardware. For example, if an accelerator supports per-chip throttling and a chip is used by one task at a time, then task-level QoS differentiation is possible.

While proactive capping addresses the availability bottleneck of power telemetry unavailability, it may become a limiting factor for power oversubscription. We have to set the jailing fraction conservatively (i.e., it may be set greater than necessary) for the open-loop control to be safe. For clusters with a high portion of latency-sensitive tasks, only a small jailing fraction may be feasible, leading to a small oversubscription. To increase oversubscription for those clusters, it may be worth investing in building a reliable secondary source of power signals, either from rack- or machine-level power sensors or from machine learning models that map resource usage to power, so that closed-loop control is still functional when the primary source, data center power meters, is unavailable. Proactive capping may be used as the last resort when both the primary and the secondary sources are unavailable.

Thunderbolt is a *reactive* system (not to be confused with "reactive capping" defined in this paper), in the sense that it reacts to riskily high data center power that is present (in the case of reactive capping) or expected (in the case of proactive capping). A more *proactive* approach, such as power-aware job scheduling and admission control, may actively balance load to avoid riskily high data center power via scheduling rather than throttling. Job scheduling and admission control are largely orthogonal and complementary to Thunderbolt and are valuable candidates for future work.

9 Related Work

This work has a similar architecture as Google's power capping for medium-voltage power planes (MVPPs) [18]. It shares many advantages of the MVPP power capping, such as fast response, priority- and QoS-awareness, platform-independence, and scalability, while making a critical improvement of not interrupting throughput-oriented workloads. This is to be contrasted with the MVPP power capping design that uses Linux SIGSTOP and SIGKILL signals. This work also introduces the proactive capping sub-system to improve system availability, which the MVPP capping system does not have.

Our primary, reactive capping subsystem uses node-level CPU bandwidth control provided by the Linux kernel's CFS scheduler. To our knowledge this is the first time this node-level mechanism, applicable on a per-task basis, is used for data center power management. There is literature [2] that discusses using CPU bandwidth control for power management of mobile devices, but not for data centers. Other node-level mechanisms used for power management include DVFS [6, 16, 24], RAPL [25], Intel node manager [14], power gating [16], and thread packing [6, 17].

Reactive capping also uses load shaping, a data center-level closed-loop control, as the the power control policy. Load shaping is implemented at one level of the power delivery "choke point" that constrains the overall power capacity. It is

simpler than multi-level controls in other large-scale implementations [14, 24, 25].

Our failover, proactive capping subsystem to mitigate the risk of power signal unavailability, uses node-level CPU affinity control. It is the same low-level mechanism as "thread packing" [6, 17], but in this work we use it only as a failover mechanism when power signals are unavailable because of its limitations compared to CPU bandwidth control.

Other studies also use power-aware job scheduling and admission control to limit power draw [4, 12, 23]. Compared to node-level and hardware-level power throttling mechanisms such as ours, these scheduler-level techniques can improve availability and performance of running jobs. It is a valuable direction for future work, as discussed in Section 8.

10 Summary

In this paper we present Thunderbolt, a throughput-optimized and QoS-aware power capping system that is robust and scalable. We elaborate important design choices and present production evaluation of its policy decisions. Thunderbolt has been deployed in warehouse-sized data centers and saved us millions of dollars on capital expenses by enabling otherwise nonexistent additional power capacity in our data centers.

Acknowledgments

This work is the result of multi-year efforts contributed by many engineers, managers, and supporting staff. We would like to thank Strata Chalup, Greg Imwalle, Tom Kennedy, Dave Landhuis, Mian Luo, Matthew Nuckolls, Pablo Perez, Etienne Perot, Brad Strand, Jeff Swenson, Jikai Tang, Steve Webster, Quincy Ye, Henry Zhao, and Steven Zhao for their contributions and support for the Thunderbolt program. We are also grateful to David Culler, Gernot Heiser, Jeff Mogul, and the anonymous reviewers for their constructive feedback.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[2] Y. Ahn and K. Chung. User-centric power management for embedded CPUs using CPU bandwidth control. *IEEE Transactions on Mobile Computing*, 15(9):2388–2397, 2016.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[4] Arka A. Bhattacharya, David Culler, Aman Kansal, Sriram Govindan, and Sriram Sankar. The need for speed and stability in data center power capping. *Sustainable Computing: Informatics and Systems*, 2013.

[5] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1):1 – 14, 1989.

[6] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & cap: Adaptive DVFS and thread packing under power caps. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–185, 2011.

[7] Intel Corporation. Power Stress and Shaping Tool. https://01.org/power-stress-and-shaping-tool. Accessed: 2020-04-15.

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

[9] Xing Fu, Xiaorui Wang, and Charles Lefurgy. How much power oversubscription is safe and allowed in data centers? In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 21–30, New York, NY, USA, 2011.

[10] Gartner, Inc. Gartner says global IT spending to decline 8% in 2020 due to impact of COVID-19. https://bit.ly/gartner-2020-05-13. Accessed: 2020-05-19.

[11] Synergy Research Group. Hyperscale operator spending on data centers up 11% in 2019 despite only modest capex growth. https://bit.ly/synergy-2020-03-24. Accessed: 2020-05-19.

[12] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 535–548. ACM, 2018.

[13] Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *HPCA '08: 14th International Conference on High-Performance Computer Architecture*, pages 123–134, 2008.

[14] Y. Li, C. R. Lefurgy, K. Rajamani, M. S. Allen-Ware, G. J. Silva, D. D. Heimsoth, S. Ghose, and O. Mutlu. A scalable priority-aware approach to managing data center server power. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 701–714, 2019.

[15] Robert Love. CPU Affinity. https://www.linuxjournal.com/article/6799. Accessed: 2020-04-15.

[16] Kai Ma and Xiaorui Wang. PGCapping: exploiting power gating for power capping and core lifetime balancing in CMPs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[17] Sherief Reda, Ryan Cochran, and Ayse K. Coskun. Adaptive power capping for servers with multithreaded workloads. *IEEE Micro*, 2012.

[18] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, and Parthasarathy Ranganathan. Data center power oversubscription with a medium voltage power plane and priority-aware capping. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, pages 497–511. ACM, 2020.

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[20] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Paul Turner, Bharata B Rao, and Nikhil Rao. CPU bandwidth control for CFS. In *Proceedings of the Linux Symposium*, pages 245–254, 2010.

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

[23] Guosai Wang, Shuhao Wang, Bing Luo, Weisong Shi, Yinghang Zhu, Wenjun Yang, Dianming Hu, Longbo Huang, Xin Jin, and Wei Xu. Increasing large-scale data center capacity by statistical power control. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[24] Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W. Keller. SHIP: A scalable hierarchical power control architecture for large-scale data centers. *IEEE Trans. Parallel Distrib. Syst.*, 23(1):168–176, 2012.

[25] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's data center-wide power management system. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 469–480, Piscataway, NJ, USA, 2016.

[26] Huazhe Zhang and Henry Hoffmann. A quantitative evaluation of the RAPL power control system. *Feedback Computing*, 2015.