Leveraging Application Classes to Save Power in Highly-Utilized Data Centers

Kostis Kaffes[*]
Stanford University
kkaffes@stanford.edu

Dragos Sbirlea
Google LLC
dragoss@google.com

Yiyan Lin
Google LLC
linyiyan@google.com

David Lo
Google LLC
davidlo@google.com

Christos Kozyrakis[†]
Stanford University
kozyraki@stanford.edu

ABSTRACT

Data center energy consumption has become an increasingly significant contributor both to greenhouse emissions and costs. To increase utilization of individual hosts and improve efficiency, most modern data centers co-locate workloads belonging to different application classes, some being latency-sensitive (LS) and others best-effort (BE) which are more tolerant to performance variation. It is therefore necessary to design mechanisms that reduce power consumption even in the resulting high-utilization environment, while preserving LS task performance. Moreover, the abundance of different workloads and the security implications of public cloud make mechanisms that rely on extensive knowledge of workload characteristics or on application-exported metrics challenging to deploy.

We present PACT, **P**er **A**pplication **C**lass **T**urbo Controller, a system that leverages two novel mechanisms to reduce power consumption even in highly-utilized data centers. By treating applications like opaque boxes that do not need to provide application-specific performance signals, the first mechanism, Turbo Control, reduces power consumption by decreasing the operating frequency and throttling only BE tasks, without affecting performance-sensitive LS tasks. We identify the shortcomings of Turbo Control and increase its effectiveness by introducing CPUJailing, a mechanism that allocates different sets of cores to LS and BE applications. We deploy PACT (Turbo Control + CPUJailing) in production at Google's data centers and demonstrate that it provides workload-agnostic power savings of 9% on average together with a 4% performance improvement for LS tasks across thousands of workloads and nodes.

[*]Kostis Kaffes was an intern at Google during this work.
[†]Christos Kozyrakis was partly at Google during this work.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SoCC '20, October 19–21, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8137-6/20/10.
https://doi.org/10.1145/3419111.3421274

ACM Reference Format:
Kostis Kaffes, Dragos Sbirlea, Yiyan Lin, David Lo, and Christos Kozyrakis. 2020. Leveraging Application Classes to Save Power in Highly-Utilized Data Centers. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3419111.3421274

1 INTRODUCTION

Policies to reduce energy consumption and greenhouse gas emissions have been adopted by most countries and industries. For example, California has passed into law an ambitious emissions curbing plan, i.e., to become carbon neutral by 2045 [3]. Data centers consume significant amounts of energy, accounting for about 3% of the global electricity consumption or for nearly 40% more than the consumption of the entire United Kingdom and their power consumption is expected to double every four years [2]. Hence, reducing data center power consumption is key to achieving local and global environmental goals. Furthermore, from a data center owner perspective, 25-40% of a data center's operating expenses are attributed to energy consumption [38] making power savings an important consideration.

These incentives have led the research community and the industry to work towards improving overall data center efficiency. The first step was to reduce non-computing related power consumption [5], e.g., cooling and power distribution overheads. Striving for even greener data centers, the focus has shifted to improving computing efficiency in two directions: increasing utilization and making servers energy-proportional.

The first general approach is to increase workload consolidation by co-locating different workloads, drive up utilization, and improve efficiency. In our setting, all workloads run as one of two classes of applications: latency sensitive

(LS) and best-effort (BE). LS jobs have tight Service-Level Objectives (SLOs), and are applications such as search engines, advertising systems, and in general user-facing services. BE jobs are throughput oriented applications with loose or no SLOs, such as offline analytics. The co-location of LS and BE jobs is challenging because interference between co-located workloads can lead to degraded performance. Due to the significant savings it produces and the work done on mitigating interference, this approach is now a common practice in the industry [11, 14, 15, 33, 42]. Systems such as PerfIso [16] and Scavenger [17] allow the co-location of LS and BE tasks and increase utilization while reducing adverse interference effects. Nevertheless, increasing utilization on its own cannot provide the power savings necessary to achieve our ambitious environmental goals; it is also necessary to further improve efficiency by reducing power consumption even at high utilization. An alternative approach is to tackle the problem head-on and make data center servers energy-proportional, i.e., scale the servers' power consumption to match their instantaneous load. The main issue is that data center servers can still consume 30-60% of maximum power even at low utilization [4]. Mechanisms such as Dynamic Voltage and Frequency Scaling (DVFS) can lower the power consumption by running workloads more slowly while processor low power states (c-states) [1] can reduce the power consumption of idle cores. However, these mechanisms produce diminishing returns in highly-utilized data centers that host consolidated workloads.

The production environment that we target at Google has the following characteristics [40] that make existing power saving frameworks ineffective:

- Each machine is shared by tens or even hundreds of different applications (Sec. 4.1).
- Cluster frameworks and operators generally do not have visibility into application performance.
- Average machine utilization is high (> 50%).
- LS tasks are untouchable as far as performance is concerned while BE tasks can tolerate performance variability.
- Cores are generally not reserved for specific applications and can be shared among multiple LS and BE tasks.

This paper shows that it is possible to achieve considerable power savings in such an environment with high utilization and an arbitrary workload mix. To achieve that, we present **PACT**, a new system that enables different frequency scaling policies for different application classes. PACT consists of two mechanisms, Turbo Control and CPUJailing. With Turbo Control we can run LS tasks at high frequency while the power savings come exclusively from slowing down performance-insensitive BE tasks. Turbo Control does not require any workload-specific knowledge and is deployable in private and public data centers. We analyze the behavior and performance of our framework and identify that core sharing between LS and BE tasks is a significant factor that limits Turbo Control's effectiveness. To alleviate this issue, we propose CPUJailing, a mechanism that improves isolation by placing LS and BE tasks on two disjoint sets of processor cores, called jails. Unlike previous approaches, CPUJailing does not require any offline profiling and can work with any set of workloads out of the box; a necessary property that makes its deployment in shared clusters feasible. PACT has been integrated with Google's cluster scheduler framework, Borg [42], and its behavior has been evaluated in production. *We show that combining the per application class frequency scaling policy together with the isolation provided by CPUJailing reduces power consumption by 9% on average, while it improves the performance of LS tasks by 4% on average.*

In summary, the key contributions of this paper are as follows:

- We exploit the performance expectations of different application classes to develop PACT and achieve significant power savings in a data center production environment without relying on application-exported metrics and without violating SLOs.
- We leverage core-level isolation techniques to make our power savings method applicable to a wide utilization range (up to 90%). Previously, such techniques were used only for performance improvement.
- We evaluate PACT in a large-scale production environment with thousands of nodes at Google and demonstrate significant power savings without affecting the latency or the throughput of LS tasks.

The rest of the paper is organized as follows. § 2 motivates the approach of per application class frequency scaling. § 3 presents the design and implementation of the Turbo Control and CPUJailing mechanisms that compose PACT. § 4 demonstrates a thorough quantitative evaluation while § 5 discusses questions raised during evaluation. Finally, § 6 presents related work.

2 MOTIVATION

2.1 Design Requirements

From Google's data centers' characteristics described in Section 1, three key design requirements arise that any power management policy and system must satisfy:

- Mechanisms used to reduce power consumption need to be **application-agnostic** and view workloads as **opaque boxes**.

- The performance of tasks should not deteriorate outside the SLO given by their application class.
- Given the high average utilization, power savings should manifest at a wide utilization range (20%-90%).

By satisfying these constraints, PACT is applicable to a real production context, with no assumptions on specific applications, or caps on the CPU usage of individual applications, or restrictions on which application runs on which machine which have been common in previous work. This is essential for using this mechanism in Google's production data centers. The rationale behind the requirements is further explained in the rest of this section.

2.2 Why consider applications as opaque boxes?

In our shared cluster setting, the challenge is to achieve power savings without affecting application performance or by affecting it within some acceptable limits. One common host-level power saving technique, DVFS, can lower the power consumption by running workloads more slowly. DVFS policies and mechanisms have become more sophisticated over time, starting from power throttling of a chip multi-processor [21], to finer-grain per-core management [37, 39], to workload- and request-aware throttling decisions [9, 10, 20, 23, 31, 45, 46]. Newer approaches that can work in high-utilization environments generally require visibility into application-exported metrics. For example, Pegasus [23] monitors application latency to determine how much power to supply to each machine while ixcp [31] uses request queue lengths to decide both how many cores an LS application requires and the frequency settings used for these cores. Techniques that enable workload co-location are susceptible to interference caused by contention on shared resources such as CPU, caches, memory, disk, and network. A large number of existing systems [7, 12, 20, 24, 29, 31, 44, 48] mitigate this harmful interference by using application-level metrics to monitor the performance of LS tasks and partition the resources among the LS and the BE tasks.

In Google's environment, application-level metrics are not available. A cloud provider does not and should not have visibility into client workloads running as containers or virtual machines on its infrastructure. Moreover, even private clusters are shared by applications and tasks belonging to different organizations and users, making it hard to export and even harder to balance resource allocation across different application-level metrics. That led to the emergence of opaque-box approaches [16, 17] that partition resources dynamically by reacting to demand changes in an online manner. Such techniques have been used to improve performance through better resource allocation. However, we know of no existing opaque-box system that can reduce data center power consumption without violating application SLOs.

2.3 Why use application classes?

In Section 2.2 we saw that any mechanism usable in shared and public data centers cannot have access to application-level metrics and SLOs. This constraint seems to set up an impossible problem; how can we avoid violating user-facing SLOs without being able to monitor them?

The key observation we make is that we can leverage the different expectations about the performance of different application classes. At Google, the performance expectations of each application are encoded in its chosen application class. This approach lets application developers choose the application class by reading the performance specifications of each class and then testing the actual performance of their application as if it were running as different classes. They use the least "expensive" class that satisfies its performance needs.

Applications running under the LS class expect consistently great performance, so we cannot slow them down to save power while maintaining a opaque-box view. On the other hand, BE tasks can expect performance swings that dwarf the performance effect of our power savings approach. BE tasks do not have a latency SLO; their throughput SLO simply ensures that they are getting the amount of CPU they request, computed across multi-minute windows. This allowed us to apply our method to BE tasks without breaking SLOs and without having to migrate sensitive apps to LS. Hence, by considering application-class SLOs instead of application-level ones [27], we simplify the problem to reducing power consumption without affecting the performance of LS tasks in general.

2.4 Why focus on high utilization?

While the constraints of highly-utilized shared environments are not amenable to the possibility of additional power savings, high utilization is now the common case in Google's production data centers [40] so we focused on exploring potential solutions for this situation. Modern data centers use power over-subscription [13, 25] which means more servers are deployed than can be supported at peak load. Consequently, enabling power savings at high machine utilization is critical since that is when the peak power requirements must be met. To fit within the available power budget, servers may have to be limited in their power usage or even shut off by power capping [6, 22, 35]. Avoiding these extremely user-impacting events leads to better user experience, can enable higher power over-subscription, and saves on capital expenses by delaying the building of new data centers.

3 PACT

To meet the aforementioned requirements, we propose **P**er **A**pplication **C**lass **T**urbo Controller (PACT), a framework that achieves power savings without sacrificing latency-sensitive application performance in a shared cloud environment, where each machine can be highly utilized and have a mix of applications belonging to different classes. The framework contains two major components:

- **Turbo Control** dynamically switches the frequency of each CPU core to match the requirements of the application currently running on that core.
- **CPUJailing** enhances Turbo Control by placing different application classes to different sets of cores.

Turbo Control and CPUJailing are analyzed in depth in Section 3.1 and Section 3.2 respectively.

3.1 Turbo Control

Turbo Control is a mechanism that sets each core's maximum operating frequency according to the class of the application running on the core at each point in time. Every time there is a context switch between applications belonging to different classes, the maximum frequency is adjusted to the one specified for the newly-scheduled application. In our setting, we have only two application classes, latency sensitive (LS) and best-effort (BE) tasks. To meet the requirements specified in Section 2, Turbo Control runs LS applications at a fixed high frequency. The policy is platform-aware and sets that frequency to be the all-core turbo frequency of each processor, i.e., the maximum frequency that all cores can run at simultaneously without exceeding the machine's power budget. We opted to use the all-core turbo frequency instead of the processor's maximum operating frequency for predictability reasons. Otherwise, on machines with low utilization, the higher CPU frequency could boost performance, but that benefit would disappear when the machine got more loaded. Such a behavior would make it harder both for the cluster-level scheduler and for human operators to provision the right amount of resources for different workloads. For BE tasks, Turbo Control uses the processor's base frequency to limit power consumption without unreasonably degrading performance. This is not a magic value; one can always select a higher frequency for less power savings and higher BE performance, or a lower frequency for more power savings and lower BE performance.

With this design, the Turbo Control component, as a standalone part, is able to achieve significant power savings at low and medium utilization. Figure 1 indicates that Turbo Control alone reduces power consumption when overall machine utilization is below 60%. However, its limitations appear at utilization higher than 60% as power consumption

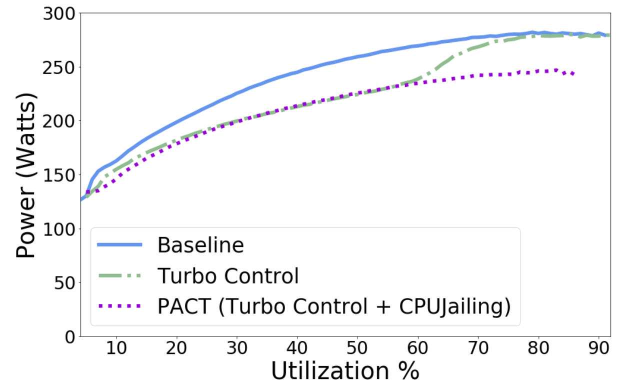


Figure 1: Average power consumption in the machines of a cluster as a function of machine utilization.

converges to the baseline. More details about the configuration and the methodology for this experiment are provided in § 4. When machine utilization is low, the Linux kernel scheduler tends to spread the load to different physical and logical (hyperthreads) cores. Thus, the probability of co-locating applications belonging to different classes on the same physical core is lower. As machine utilization increases, such overlapping is unavoidable. Additionally, the scheduler is not hyperthread-aware and it does not consider frequency requirements when taking thread placement decisions. As a result of this, the higher the utilization the more likely that two hyperthreads belonging to the same physical core run different application classes. In that case, the platform hardware corrects the conflict by running the core at the higher of the two frequency settings. This is a desirable design as it guarantees high performance for latency sensitive applications. On the other hand, it neutralizes the power saving effort when application-class overlapping is high.

Not only does the co-location of LS and BE tasks on the same physical core limit power savings but it can also affect the performance of LS tasks. Every time the execution of an LS task follows that of a BE task on the same core, the frequency increases. However, the increase does not happen instantaneously. As shown in Section 4, this frequency switching latency can result in running LS tasks at a lower frequency for short periods of time after context switching which degrades their performance and violates the requirements set in § 2.

Due to the aforementioned issues, Turbo Control by itself would not be effective in our production environment. To mitigate them, we propose a CPU isolation mechanism called CPUJailing which is described in the next section.

3.2 CPUJailing

CPUJailing is a CPU isolation mechanism that aims to separate LS and BE tasks in shared cluster environments. It requires no offline training or configuration and thus it can be directly applied to any workload combination. We showcase that, by placing LS and BE tasks in separate physical cores, CPUJailing enables Turbo Control to achieve power savings even at high utilization as shown in Figure 1. Moreover, the improved performance it provides by mitigating interference due to physical and logical core sharing and its general applicability makes it worth to be considered as a stand-alone mechanism (§5).

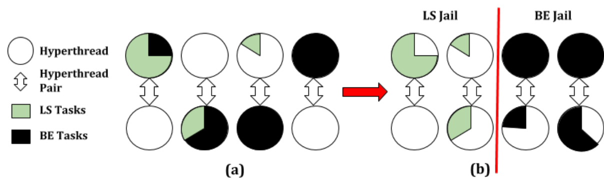


Figure 2: Core placement without (a) and with (b) CPUJailing.

3.2.1 Design

. The main component of CPUJailing is a controller that runs periodically and works on a per socket basis. Algorithm 1 describes the controller's functionality. First, the controller calculates the total CPU utilization of all LS and BE tasks in a socket (lines 5,7). Then, based on these utilization metrics, it determines whether the mechanism can be *active* in the socket. The mechanism is active when the LS and the BE CPU usage can fit in the socket along with a number of buffer cores for LS tasks, i.e., an LS buffer (line 9).

These extra cores are necessary as LS tasks can have very bursty behavior and utilization patterns [16]. With approaches that assign tasks to subsets of cores, an important risk is that LS tasks that need CPU time may not have access to it due to the subset being too restricted. If an LS task is limited to a subset of the cores of the system and such a spike happens, its inability to expand can lead to severe service level objective violations, especially regarding its tail behavior [16]. If we are pro-active and over-provision cores to the LS tasks by creating the LS buffer (line 6), we can avoid such a problem. This is demonstrated in Section 4 as we observe no LS performance regression due to CPUJailing. The sizing of the buffer is discussed in §3.2.2.

When the tasks can fit in the socket and therefore the mechanism is active, a number of cores equal to the sum of the LS utilization and the LS buffer is added to the system-wide LS CPU mask (lines 10,11). The rest of the cores in the socket are added to the BE CPU mask (lines 12,13). If high utilization prevents the LS buffer allocation, we cannot guarantee CPU isolation. CPUJailing self-disables in the socket, the full socket CPU mask is added to both the LS and BE system-wide jails (lines 15,16), and we fall back to the original core allocation.

Once this process is done for all sockets, the CPU masks of the LS and BE tasks are adjusted accordingly. More specifically, we choose to use the intersection of the CPUJailing CPU mask and the pre-existing CPU mask for each task (lines 21-26) in order to minimize the interference with other control mechanisms that might be adjusting the tasks' CPU masks. The pre-existing mask in this case is set by these other mechanisms and has nothing to do with the output of previous iterations of the CPUJailing controller. For example, we would not want to interfere with any throttling policy that might restrict the number of runnable cores for a given task. By taking the intersection between the two CPU masks, we make sure that the task is going to run on a subset of the cores it was originally going to run on.

1 CPUJailing Controller

```
1: procedure CPUJailing :
2:    ls_jail ← ∅
3:    be_jail ← ∅
4:    for socket in sockets do
5:       ls_util ← ceil(get_ls_util(socket))
6:       ls_util ← ls_util + ls_buf
7:       be_util ← ceil(get_be_util(socket))
8:       num_cores ← get_num_cores(socket)
9:       if be_util + ls_util ≤ num_cores then
10:         ls_socket_mask ←
11:            get_N_CPUs(socket, ls_util)
12:         be_socket_mask ←
13:            socket_mask \ ls_socket_mask
14:      else
15:         ls_socket_mask ← socket_mask
16:         be_socket_mask ← socket_mask
17:      ls_jail ←
18:         ls_jail ∪ ls_socket_mask
19:      be_jail ←
20:         be_jail ∪ be_socket_mask
21:   for task in ls_tasks do
22:      task.cpumask ←
23:         task.cpumask ∩ ls_jail
24:   for task in be_tasks do
25:      task.cpumask ←
26:         task.cpumask ∩ be_jail
```

3.2.2 Design Considerations

. **Sizing the LS Buffer:** We explored many options regarding how to size the LS buffer. We considered having a fixed number of cores, e.g., 4 or 8 similar to the PerfIso [16] system, or scale it dynamically according to the LS tasks' CPU utilization. Our decision was guided by the fact that we want to run BE and LS tasks on separate *physical cores* in order to enable per application class turbo control. This constraint guided our design choice. We dynamically adjust the buffer size to be equal to the LS task utilization in each socket with the buffer consisting of the hyperthread pairs of the cores belonging to the LS jail. For example, for an application that used 4 hyperthreads during the previous monitoring period, the CPUJailing controller would allocate 4 hyperthreads for the LS jail and 4 for the LS buffer, i.e., 8 hyperthreads belonging to 4 physical cores in total. This way, LS tasks run on separate physical cores from BE tasks without sacrificing their performance. Tasks that benefit from shared caches can use hyperthread pairs while single-threaded tasks or tasks that perform better if they fully occupy a physical core will not suffer from interference.

If there are not enough idle cores for the buffer, the CPUJailing mechanism self-disables. This can happen only when the utilization is high and attributed mostly to LS tasks. As we show in §4.1, in Google's data centers the job mix is balanced making this a rare occurrence. Moreover, in such a scenario there are not many opportunities for power savings anyway since LS tasks cannot be throttled.

Jail size stability: Initially, the CPUJailing controller loop ran every second and adjusted the jail sizes and CPU masks accordingly. However, we observed that this caused very frequent jail adjustments, increased CPU usage by the controller itself and led to some loss of locality for the tasks as they were bouncing around. Since we care mostly about the performance of the LS tasks, we adjusted the controller accordingly. It still checks the utilization every second and calculates new jail sizes. If the size of the LS jail increases, the controller normally assigns more cores to the jail and adjusts the CPU masks. If the size of the LS jail decreases, it does not change the jail allocation until thirty seconds have passed since the previous adjustment. This way, we avoid many marginal and unnecessary adjustments without penalizing the performance of LS tasks.

3.3 Implementation

Both Turbo Control and CPUJailing were implemented in C++ as additions to Borglet, the per-node agent of Google's cluster management system [42]. Borglet runs as a user-space daemon that manages applications isolated using Linux cgroups.

For Turbo Control, the user-space node agent interprets a policy file that specifies frequency requirements for the different application classes. It is able to determine the class of each application from information present in cgroupfs. However, the real control of core frequency is implemented via a new kernel patch. The patch leverages fine-grained scheduling control and adjusts the core frequency based on the class of the upcoming process every time the scheduler initiates a context switch. The kernel is able to limit the maximum frequency and implement the policy specified for each application class by writing the user-defined input to the processor's MSR registers.

Information necessary for CPUJailing is also read from cgroupfs. These data include but are not limited to metrics regarding resource usage, e.g., CPU utilization, various properties, e.g., priority, and limits, e.g., CPU mask. Similarly, tasks' CPU masks are updated by writing to the corresponding cpuset files in cgroupfs.

To accommodate for more application classes in the future, we can define additional operating frequencies based on each class' SLO for Turbo Control and allocate additional jails for CPUJailing.

3.4 What other approaches have we considered?

Other approaches under consideration were:

- Run all tasks at a lower frequency. This reduces LS job performance and may require more servers to handle peak load.
- Run tasks belonging to different application classes on different machine pools and apply different policies per pool, e.g., employ DVFS on the BE pool only. This makes it difficult to reach high utilization.
- Run BE tasks on a specific subset of cores in each machine. This assumes a certain distribution of LS and BE tasks which is generally unknown to the cluster owner and hard to predict [16].
- Let BE jobs run only at low utilization time. This approach is compatible with PACT .
- Observe application-level metrics and do fine-grain management. This is only possible if customers offer visibility to such metrics.

4 EVALUATION

In evaluating PACT, we aim to answer the following questions:

(1) What are the characteristics of the workloads and the evaluation environment we use? (§4.1)
(2) How much power can we save? (§4.2)
(3) Do we have to sacrifice LS performance in order to save power? (§4.3)

(4) Both components of PACT are potentially usable in separation. What would be the resulting performance and power savings?

Experimental Setup: We evaluate PACT in two production clusters at Google, Cluster A and Cluster B, each with more than 10,000 machines of different hardware platforms. Tasks submitted by different teams and organizations are placed to individual servers by centralized cluster managers. Each machine can run tens or hundreds of tasks with different characteristics and performance requirements. This setup creates a natural testbed which allows us to evaluate PACT and detect statistically significant performance improvements. We randomly assign each machine to one of 4 sets, each with a different configuration.

- **Baseline:** The pre-existing configuration that runs all workloads using the all-core turbo frequency.
- **CPUJailing:** Only the CPUJailing mechanism is enabled.
- **Turbo Control:** Only the Turbo Control mechanism is enabled.
- **PACT:** Both the Turbo Control and CPUJailing mechanisms are enabled.

Metric Collection: Performance and power metrics from each machine are collected and aggregated at 5-minute intervals by a centralized service. For our evaluation of PACT we analyze the metrics over a 10-day period in November 2019. We report the metrics in two flavors. First, bar charts are included to show the value of different metrics aggregated and averaged cluster-wide for each day of the 10-day period. The error bar on these charts shows the standard deviation across different days. Second, to explain and reason about the behavior of the mechanisms we evaluate, we plot power and performance metrics as functions of the machine utilization. We place machines in different buckets according to their CPU utilization and aggregate the metrics of interest for each bucket. We define utilization as the CPU usage over the 5-minute metric collection window. Utilization buckets where the number of samples is small enough that is not representative have been omitted. Power measurements are collected using Intel's Running Average Power Limit (RAPL).

4.1 Cluster and Workload Characteristics

In this section we explore the characteristics of the clusters that we use to evaluate PACT. Both clusters have machines with Intel CPUs that support per-core frequency adaptation; machines in Cluster A have CPUs belonging to an older generation.

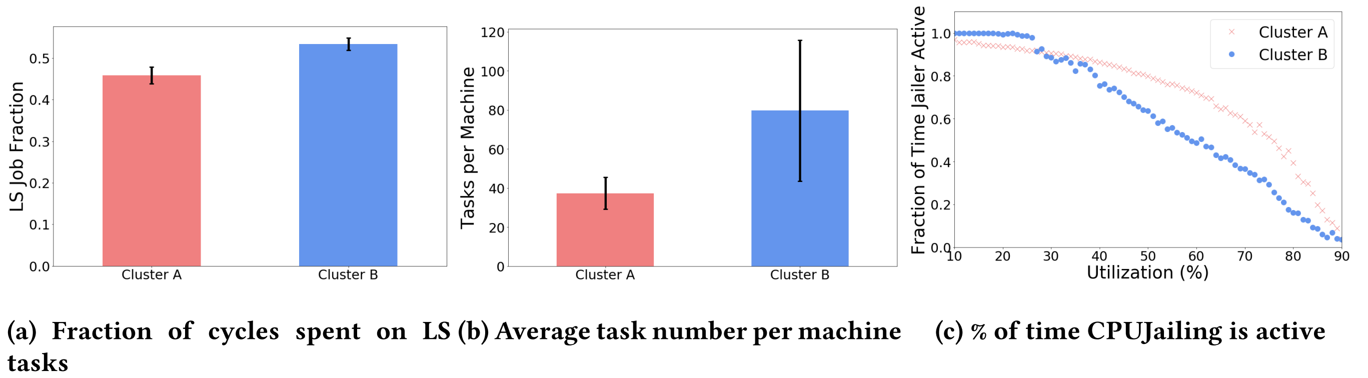
First, we compare the fraction of cycles in each cluster that are consumed by Latency Sensitive tasks. In Figure 3a we observe that Cluster A spends 45.8% of its cycles on LS tasks, while Cluster B is more LS-heavy, spending 53.4% of its cycles on LS tasks. As expected due to how Turbo Control and CPUJailing work and as we will see in §4.2, the higher fraction of LS tasks in Cluster B limits the potential power savings there.

Figure 3b compares the average number of tasks per machine on each cluster. Cluster A machines host on average 37 tasks while Cluster B machines host 80 tasks. The higher number of tasks per machine in Cluster B can be explained by the fact that it has newer CPUs with more cores. It is also worth noting the very high variability of the task count in Cluster B; the standard deviation is 36.

Figure 3c shows the fraction of time CPUJailing is active over the machine utilization for the two clusters. If there are not enough idle cores for the LS buffer allocation, CPUJailing self-deactivates in a socket. The more often CPUJailing is active, the more likely that LS and BE tasks do not run on the same physical or logical core and the more effective Turbo Control is. As expected, CPUJailing is more likely to be inactive at high utilization since there are fewer idle cores available for the LS buffer. CPUJailing is less active in Cluster B due to its larger fraction of LS tasks.

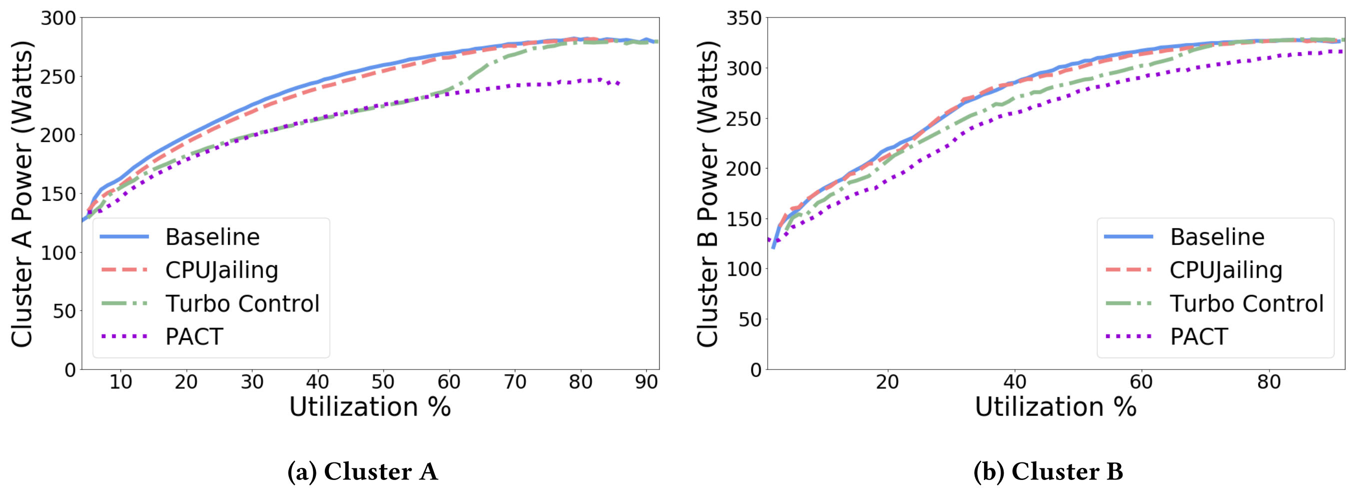
4.2 Power Savings

To evaluate the effect of PACT on power consumption, we measure the package power of each machine. The package power includes the power consumed by the CPU cores, cache, and other uncore components, e.g., the memory controller and integrated graphics processors. Figure 4 examines how the average power consumption varies as machine utilization changes. In all different deployments the power consumption goes up as utilization increases. Baseline consumes the most power in both clusters while CPUJailing is slightly below that. In Cluster A both Turbo Control and PACT achieve significant power savings at low to medium utilization (up to 60%). For utilization higher than 60% Turbo Control converges to the behavior of Baseline and CPUJailing. This is expected as the higher the utilization the more likely LS and BE tasks are co-located and therefore Turbo Control is not effective. However, PACT demonstrates significant power savings for utilization up to 85%-90% by leveraging CPUJailing to keep LS and BE tasks on separate sets of cores and allowing Turbo Control to continue to throttle BE tasks. The behavior of the mechanisms is the same in Cluster B barring two interesting differences. Both deviations can by explained by the higher ratio of LS tasks that are hosted in Cluster B. First, there is not as big an opportunity for BE task throttling as there is in Cluster A and thus the overall power savings achieved by PACT are smaller. Second, LS tasks tend to dominate the machine and get co-located with BE tasks even in non-saturated machines. Hence, Turbo Control is visibly worse than PACT even at low to medium utilization.



(a) Fraction of cycles spent on LS tasks **(b) Average task number per machine** **(c) % of time CPUJailing is active**

Figure 3: Cluster characteristics.



(a) Cluster A **(b) Cluster B**

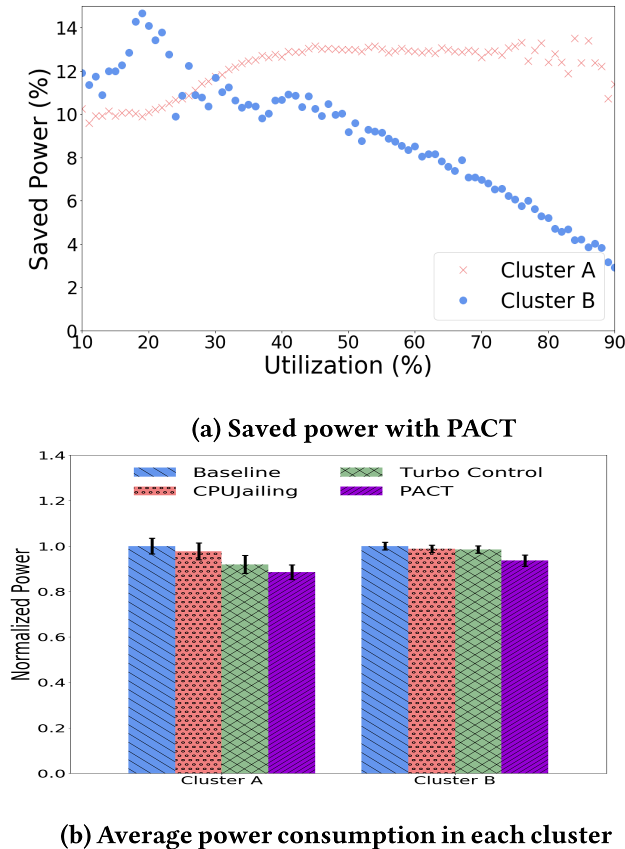
Figure 4: Power consumption as a function of machine utilization.

Figure 5a depicts the percentage-wise power savings between Baseline and PACT and reveals another difference between the two clusters that is not immediately visible from the previous charts. The power savings for Cluster A remain consistent at about 12% as utilization increases while those for Cluster B decrease. This behavior can be explained by the fact that the fraction of time CPUJailing is active decreases more drastically at higher utilization in Cluster B compared to Cluster A (Figure 3c). Next, we examine the most important metric regarding power savings, i.e., the normalized power consumption across the whole clusters (Figure 5b). *Turbo Control by itself can save about 8% and 1.6% of total power in Clusters A and B respectively while with the addition of CPUJailing the savings for PACT go up to 11.5% and 6.5%.*

We observe that CPUJailing by itself saves about 2.4% and 1.2% in the two clusters. Initially, this seems hard to explain as CPUJailing does not change any power settings. However, Figure 6 shows that these power savings can be attributed to increased low power c-state occupancy. C-states are idle power-saving states that a core enters when there is nothing to execute. C0 is the default power state of when the core is not idle. As a core's c-state number increases, it consumes less power but also has a higher wake-up latency. We observe that cores in the two configurations with CPUJailing enabled spend a larger fraction of the total time in the high C3/C6/C7 c-states as task execution is limited to a subset of the total cores and more cores are idle. The lower high c-state occupancy in Cluster B explains the reduced savings compared to Cluster A.

4.3 Performance Metrics

We evaluate the effect of PACT on application performance using two workload-independent metrics. As a **proxy for throughput** we use **instructions per core usage per second** with the goal of quantifying the progress made by the



(a) Saved power with PACT

(b) Average power consumption in each cluster

Figure 5: Power saved in the two clusters.

application. For example, if a task completed 9,000,000 instructions using 4.5 cores over the past 2 seconds, "instructions per core usage per second" would be 9,000,000 / 4.5 / 2 = 1,000,000. Figure 7 shows the average normalized throughput per application class per cluster. We observe that using Turbo Control without CPUJailing leads to a lower throughput for LS tasks by 6.5% and 1.7% for Clusters A and B respectively. However, when CPUJailing is enabled by itself, the throughput increases by 7.4% and 6.4%, while in PACT – when both Turbo Control and CPUJailing are active – we obtain more modest gains of 4.1% and 3.8%. BE task throughput decreases by 10.9% for Cluster A and 9.7% for Cluster B when both mechanisms are active. These numbers showcase the necessity of CPUJailing and validate our claim from § 2 that the frequency adjustment latency can lead to lower LS performance if left unchecked.

Next, we want to see how the performance of applications is affected if we control for frequency scaling. To achieve that, we use the **instructions-per-cycle (IPC)** metric that is relatively stable across well-behaved executions of applications, and is well anti-correlated with bad behavior caused by antagonists [47]. In Figure 8 we observe that when CPUJailing is enabled, regardless of the Turbo Control status, LS IPC increases by about 4% due to improved isolation and reduced interference. Interestingly, under PACT even the IPC of BE tasks improves by 4% and 5% in the two clusters. We suspect that this is due to the fact that BE tasks are not always cpu-bound and by increasing their clock cycle we hide high-latency events, such as LLC misses, that can stall their execution.

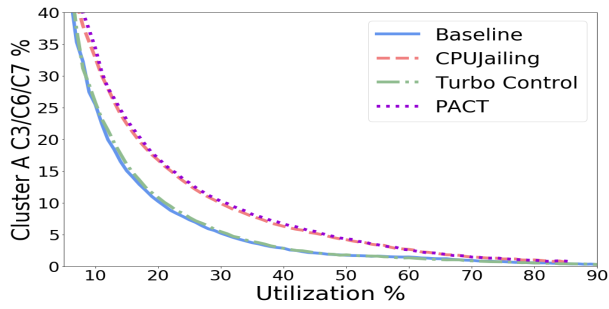
Application-level Performance: One of the constraints that we set early for PACT is that the performance of LS tasks should not deteriorate. We have shown that that is the case for the task IPC and instruction throughput. However, these metrics are not always perfectly correlated with the application performance as observed by users. To make sure that PACT does not cause any adverse effects to LS performance we conduct the following experiment. First, we select an LS application consisting of many tasks that run on a large number of machines. The selected workload is a knowledge graph serving application that has response times in the order of microseconds and runs on more than 6,000 hosts spread across the different configuration sets providing a statistical significant sample. We aggregate latency measurements and plot the cumulative distribution function (CDF) of request latency for machines that use the Baseline and PACT configurations. Figure 9 shows that the CDF lines for the two configurations overlap almost perfectly and thus proves that PACT does not affect the latency profile of LS applications.

5 DISCUSSION

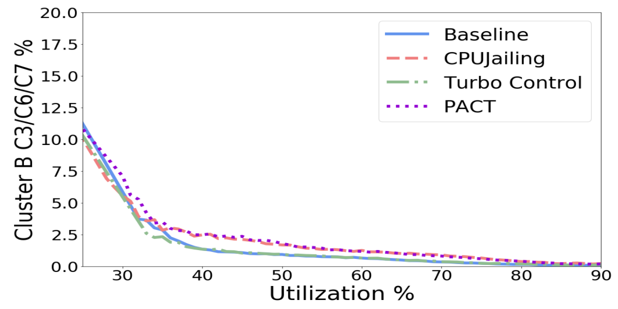
5.1 CPUJailing as a stand-alone mechanism

In Figure 7 we saw that CPUJailing by itself improves the throughput of LS tasks by about 7%. This motivates us to explore its potential usage as a stand-alone mechanism for performance improvement, similar to Scavenger [17] and PerfIso [16]. First, we need to analyze the behavior of LS tasks under CPUJailing in more detail. Figure 10 shows the LS throughput over the machine utilization for the different configurations. Generally, the throughput is higher when CPUJailing is enabled. However, there is an area at low utilization (less than 35%) where CPUJailing leads to worse performance. This behavior can be explained by the size of the buffer for LS tasks. When CPUJailing is active, the buffer size is equal to the LS utilization. LS tasks always "see" the machine they run on as if it is at 50% utilization. Thus, their throughput even for lower utilization is the same as at 50%.

There are two ways to handle this discrepancy. The first is to embrace the increased performance predictability offered by CPUJailing. The fact that the throughput is constant

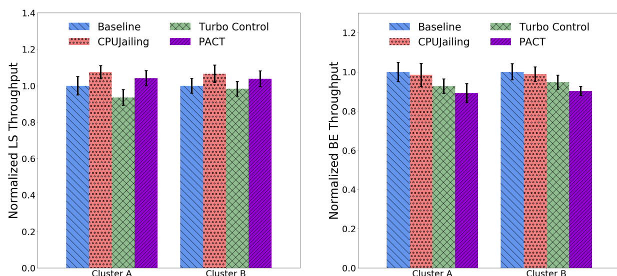


(a) Cluster A



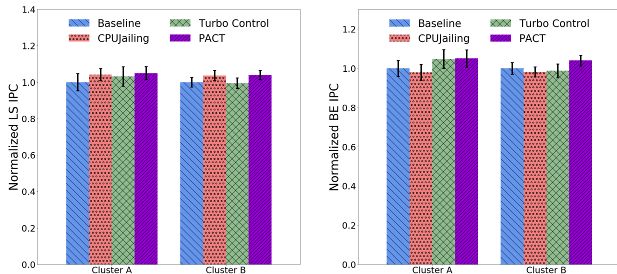
(b) Cluster B

Figure 6: C-states occupancy.



(a) LS Throughput **(b) BE Throughput**

Figure 7: Average throughput in the two clusters measured in Instructions/Core Usage/Second.



(a) LS IPC **(b) BE IPC**

Figure 8: Average Instructions-per-Cycle in the two clusters.

across different utilization levels makes it easier for administrators and automated systems to provision servers for a given workload. The other option is to provision a larger LS buffer at low utilization in order for the CPUJailing and PACT lines to match the shape of the Baseline line in the low utilization range in Figure 10.

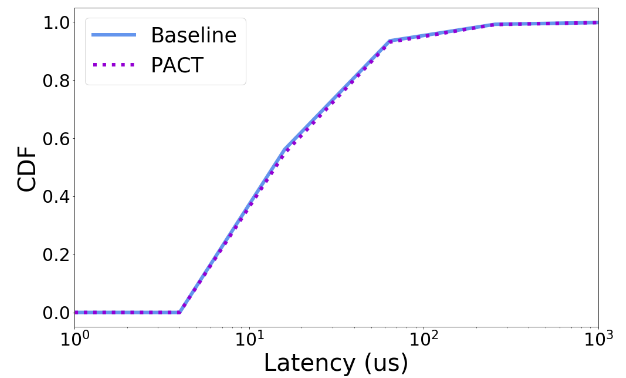


Figure 9: Latency CDF for a knowledge graph serving application across 6,000 machines.

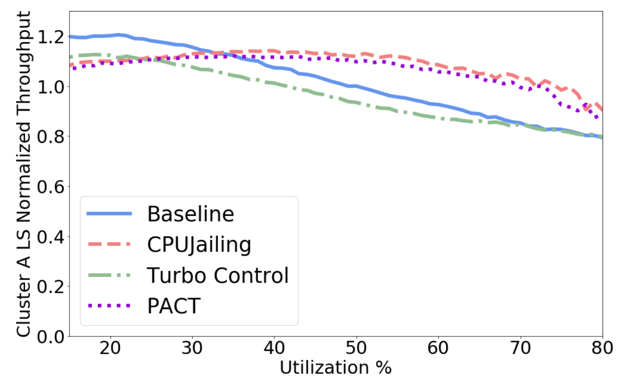


Figure 10: LS Throughput as a function of Utilization in Cluster A.

5.2 The effect of PACT on total energy consumption

As we demonstrated in Section 4.2, PACT reduces power consumption. However, it is possible that PACT achieves lower power usage with more overall energy consumption because

the throttled workloads take too long to finish execution. Below, we show that this is not the case.

We calculate the total energy consumed under PACT as a function of the energy consumed under the original configuration. We use the average throughput (in instructions per second) for both LS and BE tasks and the average power consumption for machines in cluster A. We made the choice to combine LS and BE tasks as throughput is important for LS applications as well, e.g., the requests-per-second metric for a key-value store. Moreover, the fact that very few Google applications waste cycles by busy-spinning makes instruction throughput a good proxy for application throughput and forward progress. We make the assumption that the offered load and therefore the total number of instructions for execution is the same regardless of the operating frequency.

$$\begin{aligned}
\text{Energy}_{\text{PACT}} &= \text{Power}_{\text{PACT}} * \text{Total_Execution_Time}_{\text{PACT}} \\
&= \text{Power}_{\text{PACT}} * \frac{\text{Instructions}}{\text{Throughput}_{\text{PACT}}} \\
&= 0.885 * \text{Power}_{\text{Baseline}} * \frac{\text{Instructions}}{0.96 * \text{Throughput}_{\text{Baseline}}} \\
&= 0.925 * \text{Power}_{\text{Baseline}} * \frac{\text{Instructions}}{\text{Throughput}_{\text{Baseline}}} \\
&= 0.925 * \text{Power}_{\text{Baseline}}
\end{aligned}$$

Using this model we expect the total energy consumption to be reduced by 7.75% when PACT is enabled compared to the original configuration. We must point out that this is a conservative energy savings estimate since servers are never powered off in Google's data centers and the baseline power is consumed even if the execution finishes earlier.

5.3 How would PACT compare against other power-saving approaches?

We do not directly compare PACT with prior work because no existing system could satisfy Google's requirements. For example, PerfISO [16] requires an offline training phase and access to application-level data while it does not do power management. Prior work that runs LS and BE tasks on different pools of machines would also be ineffective. In Google, machines are provisioned solely based on LS task requirements. BE tasks are used to fill the troughs in LS utilization. If we were to run different classes on different machines, we would need to provision the existing machines for LS tasks plus an additional 50% of machines (based on the fact that BE utilization is about 50% of the total) for BE tasks.

However, we believe that while the specific mechanisms we propose are Google-specific, the general idea is applicable to other environments. For example, in a setting where application-level metrics were available, one could try replacing CPUJailing with a mechanism that can leverage these additional data such as PerfIso while retaining Turbo Control and PACT's general design. Moreover, Google's requirements are common for a wide range of large scale systems; high utilization and BE/LS co-location improve efficiency while cloud providers generally do not have visibility into customers' workloads.

5.4 Future Work

Other knobs: In PACT the only knob we use to adjust the power consumption/performance trade-off for LS and BE tasks is the BE task frequency. Other obvious knobs are the LS task frequency and the relative share of cores for LS tasks versus BE tasks. We chose to use the BE frequency knob since it is the least disruptive for users; the other knobs (LS frequency or LS/BE ratio) could lead to LS performance degradation, an outcome that is unacceptable in our environment. We believe that combining different knobs while satisfying hard constraints regarding LS job performance and predicting the resulting throughput/power trade-off in a workload-agnostic manner is an open research question worth exploring.

Incorporating Application Metrics: Recent changes in Borg that allow jobs to self-report some performance metrics to the cluster manager might eventually lead to the relaxation of the opaque-box constraint [34]. We believe that the emergence of an environment where some tasks provide application-level metrics while others do not do so will result in interesting research challenges. Moreover, it is an open question whether existing approaches using application-level metrics such as Heracles [24] and Parties [7] would be effective across thousands of different workloads.

6 RELATED WORK

There are many systems with the goal of reducing power consumption or improving CPU isolation. However, as shown in Table 1 none other than PACT can meet Google's requirements, i.e., save power in a data center environment while employing a opaque-box approach. In the rest of this section we analyze in depth a series of existing power saving and performance isolation mechanisms.

6.1 Power Saving Mechanisms

There is a rich line of work on improving energy efficiency through the use of processors' power saving features. Li and Martinez [21] use DVFS to reduce the energy consumption of a single parallel workload running on a multi-core chip. They dynamically adjust the number of active processors and the chip-wide voltage and frequency levels by tracking application progress. Teodorescu and Torellas [39] and Sen and Wood [37] design power governors that operate at finer,

Mechanism	CPU Isolation	Power	Opaque-Box
PerfIso [16]	✓	✗	✗
Heracles [24]	✓	✗	✗
Pegasus [23]	✗	✓	✗
ixcp [31]	✗	✓	✗
Parties [7]	✗	✓	✗
Scavenger [17]	✓	✗	✓
PACT	✓	✓	✓

Table 1: PACT's functionality compared to existing power saving or CPU isolation approaches.

per-core scale with the goal of maximizing application performance under a specified power budget. As a metric of performance, they either use the instruction throughtput or the weighted IPC of the applications. Their approach requires extensive knowledge about the behavior and the performance of the throttled applications. Papadimitriou et al. [30] model the performance, voltage, and power characteristics of an ARM architecture. They use offline training to identify and analyze the trade-offs between energy and performance at different voltage and frequency combinations, as well as at different thread scaling and core allocation configurations.

Kanev et al. [19] bring power management to the data center era by examining the effect of c-states and DVFS on the performance and power consumption of cloud workloads. CARB [46], WASP [45], and DynSleep [10] achieve power savings through deep sleep c-states while retaining tail latency constraints. Pegasus [23] adjusts voltage and frequency at regular intervals in response to changes in the application-reported latency. Rubik [20] monitors the performance of LS workloads and uses a statistic performance model to adjust frequencies at sub-millisecond granularity to save power while meeting the target tail latency. μDPM [9] is a dynamic power management scheme that coordinates request delaying, per-core sleep states, and frequency scaling. Similar to Rubik, it requires access to application-level latency data. TailCut [8] follows a distributed approach and reduces power consumption in search engine clusters by identifying and killing stragglers. Sen and Halverson [36] and Tsirogiannis et al. [41] focus on the database domain and implement specialized power management for OLTP workloads. Finally, ixcp [31] monitors the request queue lengths of low latency networked applications and makes core assignment and frequency scaling decisions.

6.2 Isolation Mechanisms

There is significant work in systems that make feedback-based resource allocation decisions using application-level metrics. Heracles [24] uses hardware mechanisms to adjust batch tasks' resources according to the primary task's performance. Elfen [44] co-locates tasks on the same CPU only when the primary's performance is not negatively affected. TEMM [48] and RubikColoc [20] assume that the bottleneck is the last level cache or the main memory bandwidth and use DVFS and CPU share throttling to limit abusers. Quasar [12] uses machine learning techniques to infer interference patterns and make placement decisions. PARTIES [7] is a resource manager that enables the co-location of multiple LS services without QoS violations. In contrast, CPUJailing does not need application-level performance metrics or profiling. It can be combined with such approaches to act as a second-level isolation mechanism.

Scavenger [17] is one of the first systems to adopt an opaque-box approach when assigning processor resources. It uses IPC data to track the effect BE tasks have on foreground VMs and regulate CPU, last-level cache, memory capacity, and network bandwidth. CPUJailing focuses on CPU-level isolation and works in conjunction with other mechanisms that manage different resources. CPUJailing's goal is not to regulate the CPU usage of BE tasks but to make sure that they execute on different cores than the LS ones while they suffer the minimum possible performance penalty.

PerfIso [16] is the system that most closely resembles CPUJailing. It runs a single LS task on a different set of cores than BE tasks while it allocates a set of buffer cores to handle load spikes. Its main differences from CPUJailing are a) that it requires offline profiling of the LS task of interest in order to determine the buffer size and b) that it only supports the co-location of a single LS task with BE jobs. Hence, it is not suitable for shared clusters where each server may be running tens of LS tasks belonging to different users.

Another line of work uses statistical methods to model the performance of applications in shared clusters. Bubble-Up [26] and Bubble-flux [43], similarly to TEMM, focus on last-level cache and memory bandwidth interference and predict performance degradation using offline and online approaches respectively. CPI^2 [47] uses long-term CPI patterns to identify stragglers and antagonists of latency-sensitive tasks. HipsterCo [28] uses reinforcement learning to place tasks on heterogeneous systems.

Finally, there are effective but very invasive isolation solutions. Arachne [32] is a threading library that controls thread-to-core-placement and allows the co-location of latency critical and background applications with small performance impact. Shenango [29] reallocates cores across applications at very fine scale while Shinjuku [18] uses very frequent and low-overhead preemption to reduce the latency penalty of co-location. Such approaches can maximize the efficiency of data centers by allowing for almost full utilization but require extensive code changes and thus are not directly applicable to existing applications and operating systems.

7 CONCLUSION

This paper demonstrates that it is possible to achieve significant power savings through DVFS in highly-utilized data centers by leveraging the different application classes' SLOs. We introduced Turbo Control, a novel mechanism that uses per-thread dynamic frequency scaling to selectively throttle BE tasks and reduce power consumption without affecting the throughput or latency profile of LS tasks. We proposed CPUJailing as an isolation mechanism that runs LS and BE tasks in separate sets of cores, extending Turbo Control's applicability to achieve considerable power savings regardless of server utilization. PACT, combining Turbo Control with CPUJailing, evaluated across thousands of workloads and nodes in Google's production data centers, reduces cluster-wide package power consumption by 9% on average and improves the performance of LS tasks by 4% while degrading BE performance by only 10%.

ACKNOWLEDGEMENTS

We thank numerous engineers who work on the Borglet, kernel, and monitoring infrastructure teams at Google. We also want to thank John Wilkes, Xi Wang, and the anonymous reviewers for their valuable feedback.

REFERENCES

[1] 2014. Power Management States: P-States, C-States, and Package C-States. https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states.

[2] 2015. America's Data Centers Power Consumption. https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy.

[3] 2018. California Emission Curbing Plan. https://www.ca.gov/archive/gov39/wp-content/uploads/2018/09/9.10.18-Executive-Order.pdf.

[4] 2019. America's Data Centers Are Wasting Huge Amounts of Energy. https://www.nrdc.org/sites/default/files/data-center-efficiency-assessment-IB.pdf.

[5] 2019. Google Data Center Efficiency. https://www.google.com/about/datacenters/efficiency/.

[6] 2019. P Power Capping and HP Dynamic Power Capping for ProLiant servers. https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c01549455.

[7] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, Providence, RI, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[8] Chih-Hsun Chou, Laxmi N. Bhuyan, and Shaolei Ren. 2017. TailCut: Power Reduction under Quality and Latency Constraints in Distributed Search Systems. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (2017), 1465–1475.

[9] Chih-Hsun Chou, Laxmi N. Bhuyan, and Daniel Wong. 2019. μDPM: Dynamic Power Management for the Microsecond Era. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2019), 120–132.

[10] Chih-Hsun Chou, Daniel Wong, and Laxmi N. Bhuyan. 2016. DynSleep: Fine-grained Power Management for a Latency-Critical Data Center Application. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED '16)*. ACM, San Francisco Airport, CA, USA, 212–217. https://doi.org/10.1145/2934583.2934616

[11] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. 2019. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 177–192. https://www.usenix.org/conference/nsdi19/presentation/curino

[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* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. ACM, 127–144. https://doi.org/10.1145/2541940.2541941

[13] Xing Fu, Xiaorui Wang, and Charles Lefurgy. 2011. How Much Power Oversubscription is Safe and Allowed in Data Centers. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*. ACM, Karlsruhe, Germany, 21–30. https://doi.org/10.1145/1998582.1998589

[14] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. 2019. Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. ACM, 233–245. https://doi.org/10.1145/3357223.3362724

[15] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center.

In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Boston, MA, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488

[16] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. 2018. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 519–532. https://www.usenix.org/conference/atc18/presentation/iorgulescu

[17] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. 2019. Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. ACM, Santa Cruz, CA, USA, 272–285. https://doi.org/10.1145/3357223.3362734

[18] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[19] Svilen Kanev, Kim M. Hazelwood, Gu-Yeon Wei, and David M. Brooks. 2014. Tradeoffs between power management and tail latency in warehouse-scale applications. *2014 IEEE International Symposium on Workload Characterization (IISWC)* (2014), 31–40.

[20] 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* (Waikiki, Hawaii) *(MICRO-48)*. ACM, 598–610. https://doi.org/10.1145/2830772.2830797

[21] Jian Li and José F. Martínez. 2006. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* (2006), 77–87.

[22] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sree Kodakara, David Lo, and Partha Ranganathan. 2020. Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Banff, Alberta. https://www.usenix.org/conference/osdi20/presentation/li-shaohong

[23] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 301–312. http://dl.acm.org.stanford.idm.oclc.org/citation.cfm?id=2665671.2665718

[24] 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*.

[25] Sulav Malla and Ken Christensen. 2020. The effect of server energy proportionality on data center power oversubscription. *Future Generation Computer Systems* 104 (2020), 119 – 130. https://doi.org/10.1016/j.future.2019.10.021

[26] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. ACM, 248–259. https://doi.org/10.1145/2155620.2155650

[27] Jeffrey C. Mogul and John Wilkes. 2019. Nines Are Not Enough: Meaningful Metrics for Clouds. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. ACM, 136–141. https://doi.org/10.1145/3317550.3321432

[28] 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. https://doi.org/10.1145/3144168

[29] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[30] George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. 2017. Harnessing Voltage Margins for Energy Efficiency in Multicore CPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. ACM, Cambridge, Massachusetts, 503–516. https://doi.org/10.1145/3123939.3124537

[31] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. 2015. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, Kohala Coast, Hawaii, 342–355. https://doi.org/10.1145/2806777.2806848

[32] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[33] Maria A Rodriguez and Rajkumar Buyya. 2019. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience* 49, 5 (2019), 698–719.

[34] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, Heraklion, Greece, Article 16, 16 pages. https://doi.org/10.1145/3342195.3387524

[35] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, and Parthasarathy Ranganathan. 2020. Data Center Power Oversubscription with a Medium Voltage Power Plane and Priority-Aware Capping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 497–511. https://doi.org/10.1145/3373376.3378533

[36] Rathijit Sen and Alan Halverson. 2017. Frequency governors for cloud database OLTP workloads. *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)* (2017), 1–6.

[37] Rathijit Sen and David A. Wood. 2017. Pareto Governors for Energy-Optimal Computing. *ACM Trans. Archit. Code Optim.* 14, 1, Article 6 (March 2017), 25 pages. https://doi.org/10.1145/3046682

[38] S. Singh, A. Swaroop, A. Kumar, and Anamika. 2016. A survey on techniques to achieve energy efficiency in cloud computing. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*. 1281–1285. https://doi.org/10.1109/CCAA.2016.7813915

[39] Radu Teodorescu and Josep Torrellas. 2008. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 363–374. https://doi.org/10.1109/ISCA.2008.40

[40] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[41] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah. 2010. Analyzing the Energy Efficiency of a Database Server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, Indianapolis, Indiana, USA, 231–242. https://doi.org/10.1145/1807167.1807194

[42] 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 (EuroSys)*. Bordeaux, France.

[43] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, 607–618. https://doi.org/10.1145/2485922.2485974

[44] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 309–322. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yang

[45] Fan Yao, Jingxin Wu, Suresh Subramaniam, and Guru Venkataramani. 2017. WASP: Workload Adaptive Energy-Latency Optimization in Server Farms Using Server Low-Power States. *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)* (2017), 171–178.

[46] Xin Zhan, Reza Azimi, Svilen Kanev, David M. Brooks, and Sherief Reda. 2017. CARB: A C-State Power Management Arbiter for Latency-Critical Workloads. *IEEE Computer Architecture Letters* 16 (2017), 6–9.

[47] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI^2:

CPU performance isolation for shared compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, 379–391. http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Zhang_2.pdf

[48] X. Zhang, R. Zhong, S. Dwarkadas, and K. Shen. 2012. A Flexible Framework for Throttling-Enabled Multicore Management (TEMM). In *2012 41st International Conference on Parallel Processing(ICPP)*. Pittsburgh, PA, 389–398. https://doi.org/10.1109/ICPP.2012.8