PowerChief: Intelligent Power Allocation for Multi-Stage Applications to Improve Responsiveness on Power Constrained CMP

Hailong Yang[†1] Quan Chen[◇1] Moeiz Riaz[⋆] Zhongzhi Luan[†] Lingjia Tang[⋆] Jason Mars[⋆]

School of Computer Science and Engineering, Beihang University[†]
Department of Computer Science and Engineering, Shanghai Jiao Tong University[◇]
Department of Computer Science, University of Michigan - Ann Arbor[⋆]
{hailong.yang,zhongzhi.luan}@buaa.edu.cn,chen-quan@cs.sjtu.edu.cn,{moeizr,lingjia,profmars}@umich.edu

ABSTRACT

Modern user facing applications consist of multiple processing stages with a number of service instances in each stage. The latency profile of these multi-stage applications is intrinsically variable, making it challenging to provide satisfactory responsiveness. Given a limited power budget, improving the end-to-end latency requires intelligently boosting the bottleneck service across stages using multiple boosting techniques. However, prior work fail to acknowledge the multi-stage nature of user-facing applications and perform poorly in improving responsiveness on power constrained CMP, as they are unable to accurately identify bottleneck service and apply the boosting techniques adaptively.

In this paper, we present PowerChief, a runtime framework that 1) provides joint design of service and query to monitor the latency statistics across service stages and accurately identifies the bottleneck service during runtime; 2) adaptively chooses the boosting technique to accelerate the bottleneck service with improved responsiveness; 3) dynamically reallocates the constrained power budget across service stages to accommodate the chosen boosting technique. Evaluated with real world multi-stage applications, PowerChief improves the average latency by 20.3× and 32.4× (99% tail latency by 13.3× and 19.4×) for Sirius and Natural Language Processing applications respectively compared to stage-agnostic power allocation. In addition, for the given QoS target, PowerChief reduces the power consumption of Sirius and Web Search applications by 23% and 33% respectively over prior work.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*; • **Hardware** → Power and energy;

KEYWORDS

Multi-Stage Application, Power Constrained CMP, Intelligent Service Boosting

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06. . . $15.00
https://doi.org/10.1145/3079856.3080224

ACM Reference format:
Hailong Yang[†1] Quan Chen[◇1] Moeiz Riaz[⋆] Zhongzhi Luan[†] Lingjia Tang[⋆] Jason Mars[⋆] School of Computer Science and Engineering, Beihang University[†] Department of Computer Science and Engineering, Shanghai Jiao Tong University[◇] Department of Computer Science, University of Michigan - Ann Arbor[⋆] . 2017. PowerChief: Intelligent Power Allocation for Multi-Stage Applications to Improve Responsiveness on Power Constrained CMP. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages. https://doi.org/10.1145/3079856.3080224

1 INTRODUCTION

Mitigating response latency for cloud services is critical to provide satisfactory user experience [4, 14]. Several studies show that slightly increased response latency leads to significant revenue drop for cloud service providers [8, 13, 49]. Although tremendous research efforts have been devoted to addressing this problem from various aspects such as heterogeneity [30, 36, 37] and interference [15, 16, 31, 38, 55, 56, 61, 62], it is still largely an unsolved problem as more impacting factors are continuing to be discovered.

The power over-provisioning and associated power constraint in datacenters exacerbate the latency challenges [20, 41, 59]. Contemporary datacenter consumes tens of megawatts of power and thus is not sustainable with increasing demand from emerging applications [4, 6]. Since most datacenters adopt the commodity CMP servers, it is important to improve end-to-end latency on power constrained CMP, which is particularly challenging due to the unpredictable user access pattern and the complexity to accelerate the slow queries through managing the limited power budget [39–41].

Recent work [22, 34, 35] have proposed techniques leveraging fine grained power management [45] to guarantee the service level objective (SLO) with improved energy efficiency. Based on precisely pinpointing the opportunity to trade off latency headroom with power consumption, energy efficiency can be improved without violating the SLO. However, prior techniques are applied to the applications with single processing stage, ignoring applications composed of multiple processing stages. These multi-stage applications pose new challenges to mitigate response latency within the power constraint due to their distinct characteristics across stages.

Many cloud applications including traditional Web Search [2] and emerging intelligent personal assistant (IPA) [21] commonly leverage multiple stages to process user facing queries. The definition of stage naturally fits into the processing pipeline of the application.

[1]Work was conducted as a postdoc fellow of ClarityLab at the University of Michigan.

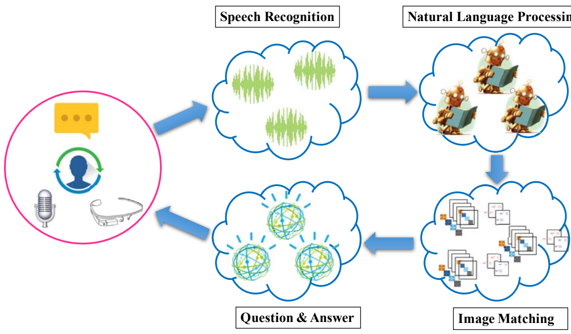

Figure 1: The processing stages of IPA applications.

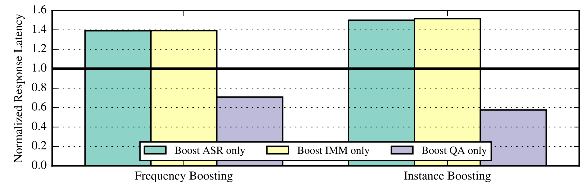

Figure 2: Normalized response latency of Sirius application when boosting different service stages.

For instance, as shown in Figure 1 a query to an IPA application flows through Automatic Speech Recognition (ASR) [23, 46], Natural Language Processing (NLP) [11], Image Matching (IMM) [7] and Question-Answering (QA) [50] stages to generate an intelligent response. To sustain the large amount of user queries, each stage consists of multiple service instances to alleviate the load. The latency at each stage is intrinsically different depending on its runtime characteristics as well as the user input [33, 60]. The result of not taking into consideration of latency variation among multiple stages with prior work [22, 34, 35] leads to several shortcomings that significantly diminish the effectiveness in mitigating response latency within the power constraint. These shortcomings include:

(1) **Unware of inter-stage bottlenecks -** Prior work assume single processing stage within an application and fail to acknowledge the intrinsic latency variance across multiple stages, which prevents effectively identifying the bottleneck service to boost throughout the query processing.
(2) **Unable to adapt stage sensitivity to various boosting techniques -** Prior work commonly adopt a particular service boosting technique during the entire execution without considering the latency sensitivity of each stage to various boosting techniques, and thus miss the opportunity to adaptively switch to the boosting technique delivering better latency improvement.
(3) **Limited power management -** Existing work managing power allocation of single stage application, fail to consider the scenario of multi-stage application, and thus are unable to intelligently manipulate power allocation across stages to accelerate the bottleneck service without violating the power constraint.

To illustrate how different boosting decisions and techniques affect response latency, we show a real world example with Sirius application [21] (details in Section 5). By boosting different service instances across stages with frequency boosting and instance boosting, while maintaining the same power budget, the response latency of Sirius application varies significantly as shown in Figure 2. The nonoptimal boosting decision (e.g., instance boosting the IMM service) results in significant performance degradation under the same power constraint. Compared to the optimal boosting decision with the right boosting technique (e.g., instance boosting the QA service), the latency reduction is more than 40%. Therefore it is critical to intelligently allocate the power budget and choose the appropriate boosting technique to improve responsiveness of multi-stage applications on power constrained CMP.

Based on these observations, we propose *PowerChief*, a runtime framework that mitigates the response latency of multi-stage applications through intelligently managing the power allocation and boosting technique to accelerate the bottleneck services under the power constraint. It leverages a joint design of service and query to precisely monitor the latency statistics of each service during runtime. Through analyzing the latency statistics, the bottleneck service is accurately identified based on both historical and realtime information. PowerChief proposes a boosting decision engine to adaptively select the boosting technique that delivers better latency improvement to the bottleneck service. The boosting decision then drives the power reallocation mechanism to dynamically redistribute the limited power budget to perform the boosting technique. With the accurate bottleneck identification, adaptive boosting decision engine and dynamic power reallocation, we are able to effectively mitigate the response latency for multi-stage applications without violating the power constraint.

This paper explores a new design space for efficient runtime framework to mitigate response latency for multi-stage applications on power constrained CMP. Specifically, this paper makes the following contributions:

- **Comprehensive analysis of the obstacles for effective latency mitigation of multi-stage application -** Our investigation shows that the lack of consideration of the latency behavior of each service instance across processing stages prevents more intelligent service boosting and power allocation. This observation motivates our runtime framework design to mitigate response latency for multi-stage applications under the power constraint.
- **Accurate bottleneck identification for service instance across stages -** We propose a service and query joint design to enable monitoring the latency statistics of service instances across stages. Based on the latency statistics, we present a bottleneck identification method utilizing both historical and realtime latency metric to accurately recognize the bottleneck service during runtime.
- **Adaptive boosting decision engine for optimal boosting technique -** We present an adaptive boosting decision engine that estimates the latency improvement of different boosting techniques and selects the one with better latency improvement to accelerate the bottleneck services.

- **Dynamic power reallocation for accommodating the boosting decision -** We design a dynamic power reallocation mechanism to recycle the power budget from non-bottleneck services, and provide the corresponding power to accommodate the boosting technique applied to accelerate the bottleneck services.

Our evaluation on a real system deployment of Sirius and NLP applications demonstrates that PowerChief is able to mitigate the response latency of Sirius and NLP applications by 20.3× and 32.4× respectively over stage-agnostic power allocation, and 99% tail latency by 13.3× and 19.4×. In addition, for Sirius and Web Search applications with a given QoS target, PowerChief saves more power compared to existing work by 23% and 33% respectively.

2 UNDERSTANDING RESPONSE LATENCY OF MULTI-STAGE APPLICATION

In contrast to applications with single processing stage, multi-stage applications exhibit intrinsic latency variance across processing stages. Thus it is more susceptible to long response latency. For the investigation of the response latency of multi-stage application under the power constraint, we aim to answer the following questions:

(1) What is unique about multi-stage applications that prevents prior work to be effective?
(2) Why is it less optimal to statically select a boosting technique to accelerate the processing stages?
(3) What is the difficulty in acquiring enough power performing service boosting under the power constraint?

2.1 Multi-Stage Application

Multiple processing stages are commonly used in nowadays user facing applications, where each stage implements an unique processing logic to understand the user query and generate desirable responses. To scale out, varying number of service instances are launched within a single stage to process queries simultaneously as shown in Figure 3. Each service instance is running on an individual processor core and maintains its own queue structure to smooth load burst. In the meanwhile, each service instance can adjust its processing speed through manipulating the core frequency.

Given a power budget, it is extremely challenging to achieve an optimal power allocation to setup the number of service instances within each stage as well as the processing speed of each service instance to mitigate response latency. Even if the optimal power allocation can be found through exhaustive search, the undetermined runtime factors such as load burst easily generate dynamic bottlenecks at potentially any service instance, which undermines the effectiveness of the static power allocation.

2.2 Difficult to identify bottleneck service

Since bottleneck services dominate the processing delay and contributes to the response latency, it is more effective to boost the bottleneck service using the limited power budget. However, accurately identifying the bottleneck service across multiple stages relies on considering latency statistics that are constantly changing during runtime. For example, service instance I_a^1 as shown in Figure 3 has a longer queue length than service instance I_b^2, which indicates instance

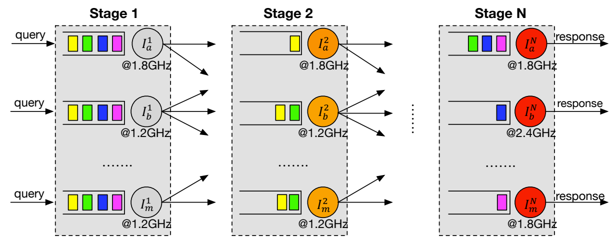

Figure 3: The exemplified setup of a multi-stage application.

I_a^1 has more chance to become a bottleneck for the future queries. However, if considering the processing speed, instance I_a^1 may run at a higher frequency that processes queries much faster than instance I_b^2. It is possible that instance I_a^1 finishes the queued up queries earlier than instance I_b^2, which reversely leaves instance I_b^2 as the future bottleneck.

Unfortunately, existing work [22, 34, 35] fail to acknowledge applications with multiple stages through the processing of a query, and are missing support to monitor the latency statistics of each service instance across stages during runtime. In addition, prior approach lack the ability to perform accurate bottleneck identification to reduce response latency within limited power budget.

2.3 No Single Boosting Technique Always Wins

Although multiple boosting techniques exist to accelerate the bottleneck service, they may achieve quite different latency improvement under the same power budget. For example, frequency boosting which increases the processing speed of the service instance is more beneficial when the serving delay dominates the latency. Whereas, instance boosting which launches new instances to share the load of the bottleneck service improves the latency better when there is a burst of queries.

The varying latency benefit of applying different boosting techniques is illustrated in Figure 4 with Sirius application. During the low load, frequency boosting improves the average and 99% percentile latency by 1.46× and 1.41× respectively over baseline, however instance boosting only achieves 1.20× (average) and 1.04× (99% percentile). Whereas during the high load, instance boosting improves the average and 99% percentile latency by 25.11× and 14.77× compared to 1.82× and 1.96× achieved by frequency boosting due to the dominate queuing delay [28]. Considering the varying load commonly seen in user facing applications [4, 14], statically adopting a particular boosting technique fails to deliver the most latency improvement. Especially under the power constraint, it is more desirable to adaptively switch to the boosting technique that achieves better latency improvement.

2.4 Non-trivial to acquire boosting power under constraint

Under a predefined power budget, boosting the bottleneck service requires recycling power from existing service instances in order to reallocate enough power to perform the boosting technique. However, choosing the appropriate service instances to recycle the power could be non-trivial since all the service instances may affect the response

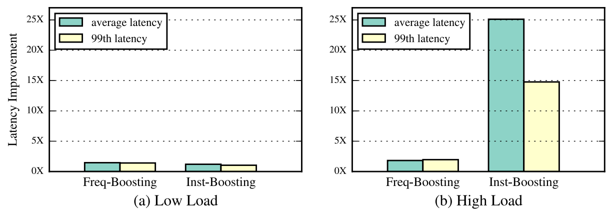

Figure 4: Varying latency improvement using frequency and instance boosting for Sirius application.

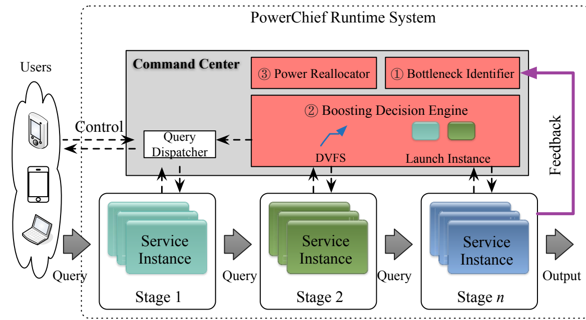

Figure 5: The Overview of PowerChief Runtime Framework.

latency as query proceeds through the service stages. Recklessly picking a service instance may cause negative effect to the response latency and diminish the latency improvement from service boosting. In the extreme case, the bottleneck could be bouncing between the service instance being boosted and the service instance whose power being recycled.

In addition, how to recycle the power also needs a careful consideration. For example, one way to recycle the power is to decrease the processing frequency of the service instance. The other one is to withdraw the service instance as long as it is not the only instance within its service stage. The latter usually recycle more power, however, it requires meticulous redirection of the query load originally sent to the service instance withdrew. Otherwise, it may generate unexpected long queue at the service instance receiving this additional load. It is critical to design an effective power reallocating mechanism to boost the bottleneck service within limited power budget.

2.5 Summary

The take-aways from our investigation on mitigating response latency of multi-stage application under the power constraint are summarized as follows:

- **Bottleneck identification requires awareness of latency variance of service instances across stages -** There is lacking acknowledgement of intrinsic latency variance of service instances of multi-stage application by prior work, which prevents accurate bottleneck identification during runtime.
- **Latency improvement varies using different boosting techniques -** Different boosting techniques have their own sweet point in accelerating the bottleneck service. Adaptively selecting the boosting technique that delivers better latency reduction significantly improves the boosting efficiency within limited power budget.
- **Power reallocation requires careful design to support service boosting -** Reckless power reallocation of limited power budget diminishes the latency improvement from bottleneck boosting. Choosing the appropriate service instance as well as the way to reallocate power budget requires a carefully designed mechanism.
- **A runtime framework is required for mitigating response latency of multi-stage application under the power constraint -** There are three critical capabilities the runtime framework needs to possess: *1)* the accurate identification of the bottleneck service during runtime, *2)* the adaptive selection of boosting techniques to achieve better latency improvement, *3)* the effective reallocation of the power budget to accelerate the bottleneck service.

Based on these findings, we propose PowerChief, a runtime framework of intelligent power allocation for multi-stage applications to improve responsiveness on power constrained CMP, through accurately identifying the bottleneck service and adaptively applying the boosting techniques during runtime.

3 POWERCHIEF FRAMEWORK

This section describes the design of the PowerChief runtime framework, which takes advantage of accurate bottleneck identification to pinpoint the service instance that contributes to the long response latency across stages, and adaptively applies the appropriate boosting technique that accelerates the bottleneck service with power carefully recycled during runtime.

PowerChief Overview The overview of the PowerChief runtime framework is presented in Figure 5, which is composed of a *Command Center* and a *Service Instance Pool* per processing stage. When a multi-stage application is implemented using PowerChief, it registers the stage layout with *Command Center*. Service instances across stages can run in distributed way and communicate with command center as well as each other through remote procedure call (RPC). Each service instance is augmented with the ability to record the queuing and serving time it spent in processing the queries. These latency statistics are carried along by the queries as they flow through the processing stages, and finally reports to the command center. This joint design of service instance and query enables the command center to concisely monitor the latency statistics across service stages during runtime with minimum overhead, which fills in the gap of missing support to reason about the latency distribution for multi-stage application from prior work.

With the latency statistics collected from each service instance, there are three core components within the command center to perform the latency mitigation under the power constraint: *Bottleneck Identifier*, *Boosting Decision Engine* and *Power Reallocator*. Based on the determined latency metrics, the bottleneck identifier analyzes the latency statistics from each service instance and recognize the bottleneck service across stages (Section 4). This bottleneck identification is then used to drive the boosting decision engine to select the appropriate boosting technique that accelerates the bottleneck service

(Section 5). Given the boosting decision, the power reallocator carefully recycles the required power from existing service instances to perform the chosen boosting technique (Section 6). These three core components working in concert give PowerChief the capability to effectively mitigate response latency of multi-stage application under the power constraint.

4 BOTTLENECK SERVICE IDENTIFICATION METHOD

The purpose of the bottleneck identification is to accurately recognize the service instance that dominates the response latency with minimum overhead. Latency statistics of each service instance need to be collected to facilitate the bottleneck identification across stages. In addition, various latency metrics can be used to measure the delay of query processing at each service instance. Choosing the appropriate latency metrics significantly affects the accuracy for bottleneck identification.

4.1 Monitoring Latency Statistics

In order to monitor the latency statistics of each service instance during runtime, we extend the query data structure to store the latency statistics as it walks through each service instance. Correspondingly, each service is augmented with the timing ability to measure the time each query spent on queuing and processing. This service and query joint design eliminates the large amount of communications between service instances and the command center, especially when deployed in large scale environment. In the meanwhile, all the latency statistics are calculated on each service locally, there is no requirement for global clock synchronization or special hardware support. Moreover, the service instance within a stage can scale out without affecting the effectiveness of latency monitoring.

As illustrated in Figure 6, when a service instance finishes processing a query, it appends latency statistics, including instance signature (ID), the queuing and processing time, to the extended query data structure. The query carries along the latency statistics as it finishes through all the service stages. After the query completes the last stage of the processing pipeline, these latency statistics are sent to the command center. The bottleneck identifier then calculates the latency metrics such as average and 99% percentile queuing and serving delay of each service instance using the latency statistics, which is then used to drive the bottleneck identification. Compared to other latency monitoring techniques that require OS modification [51, 52] and deployment of monitoring software [53], our design is easily adopted on commodity CMP servers where frameworks [3, 19] are commonly used to transform existing applications into services. The proposed joint design can be a small add-on to the frameworks, and thus saves the burden for OS modification and special software deployment.

4.2 Identifying Bottleneck Service

Several latency metrics can be used to guide the bottleneck identification. Table 1 lists the commonly used latency metrics that are available for each service instance. However, a significant drawback of the listed latency metrics to accurately indicate the bottleneck service is that they only present the historical processing ability of the service instance without considering its current load. For example, considering the latency metric of *average processing delay*, a higher number of the metric may not always indicate the actual bottleneck. Because queries may queue up at the service instance due to the burst of load. In that case, even the service instance with lower *average processing delay* is likely to become the bottleneck.

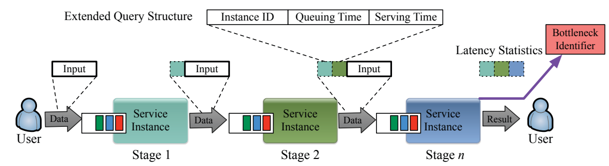

Figure 6: Service and query joint design to monitor latency statistics of each service instance across stages.

Table 1: Metrics available to identify bottleneck service

Metric	Calculation
Average queuing time	q_i
Average serving time	s_i
Average processing delay	q_i+s_i
99th queuing time	tq_i
99th serving time	ts_i
99th processing delay	tq_i+ts_i

In PowerChief, we combine both the historical latency statistics and the realtime load status to derive a latency metric that accurately indicates the bottleneck service. As shown in Equation 1, except for the average processing delay, PowerChief takes into account the real time queue length of the service instance when it performs the bottleneck identification. We use q_i, s_i and L_i to denote the average queuing time and serving time as well as realtime queue length of service instance I_i respectively.

$$LatencyMetric = L_i \times q_i + s_i \quad (1)$$

The latency metric as shown in Equation 1 can be considered as the processing delay that the incoming queries would expect since the service instance has to process the queries already in the queue before it gets back to the incoming ones. Equation 1 estimates the expected delay considering historical queuing and serving latency of of the service instance as well as the current queue length. PowerChief leverages a moving time window to calculate this latency metric for each service instance, and the one with the largest latency metric is identified as the bottleneck instance.

5 BOOSTING DECISION ENGINE

To adaptively select the boosting technique that has higher impact on reducing response latency, the boosting decision engine needs to quantitatively estimate the latency improvement of different boosting techniques without actually applying them. Moreover, different boosting techniques alleviate the bottleneck service from various aspects such as mitigating queuing and serving delay, which needs to be considered during evaluating the boosting decision. In this section, we

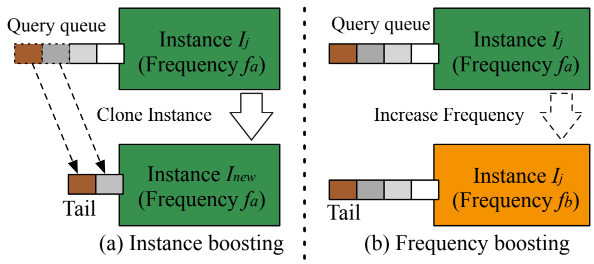

Figure 7: Mechanism of instance and frequency boosting.

first describe the estimation of latency improvement with two commonly used boosting techniques: *instance-boosting* and *frequency boosting*. We use the term *expected delay* to present the processing delay of the service instance after applying the boosting techniques. Then, we present the policy within the boosting decision engine to adaptively select the boosting technique during runtime (*adaptive boosting*).

5.1 Instance boosting

Instance boosting launches a new instance when a bottleneck service is identified. The latency improvement is achieved through alleviating the load of the bottleneck service in the current (*through work stealing*) and future form (*through load balance*). In essence, instance boosting accelerates the bottleneck service by reducing its queuing time.

In PowerChief, the new instance clones the frequency setting of the bottleneck instance as well as shares half of its load. As shown in Figure 7 (a), after a new instance I_{new} is cloned to boost the bottleneck instance I_j, half of the queries queued at the bottleneck instance I_j is offloaded to I_{new}. The estimation of the latency improvement with instance boosting can be formalized as follows.

Let q_j and s_j denote the average queuing and serving time of queries at the bottleneck instance I_j, and L_j denotes the realtime queue length of I_j. The delay of future queries at instance I_j is the time when the last query in the queue of I_j is processed. If I_j is not boosted, its processing delay can be calculated as $\left(L_j - 1\right) \times \left(q_j + s_j\right) + s_j$, where $\left(L_j - 1\right) \times \left(q_j + s_j\right)$ is the queuing time of the last query at I_j. When applying the instance boosting policy, half of the queued up queries are offloaded from I_j to I_{new}, the queuing delay of I_j is reduced to by half. Equation 2 calculates the expected delay after launching a new instance of bottleneck instance I_j. In this equation, the serving time does not change because instance boosting does not affect the processing speed of I_j.

$$T_{inst} = \frac{\left(L_j - 1\right) \times \left(q_j + s_j\right)}{2} + s_j \tag{2}$$

5.2 Frequency boosting

Different from instance boosting, frequency boosting increases the core frequency to speedup the processing of the bottleneck service as shown in Figure 7 (b). Apparently the reduction of serving time depends on the characteristics of the bottleneck service. In the meanwhile, the queuing time decreases correspondingly due to the speedup of serving.

In PowerChief, we leverage the fine grained adjustable CPU frequency on Intel Haswell architecture, providing a wide range of frequencies to boost the bottleneck service. We use offline profiling to acquire the latency reduction of the each service at different frequencies, which is then used during runtime to estimate the latency improvement with frequency boosting. The Haswell architecture adopts on-chip voltage regulators that generate sub-μs delays when adjusting the frequency [9, 28]. PowerChief takes advantage of this fast frequency transition to support queries whose QoS even at millisecond granularity.

To compare the latency improvement with instance boosting under the same power budget, PowerChief estimates the latency improvement of frequency boosting with the frequency level equivalent to the power consumption of instance boosting. The expected delay is calculated in a similar way to instance boosting. Suppose the bottleneck instance I_j and its frequency is increased from f_l to f_h, the ratio of latency reduction is α_{lh} from the offline profiling. Equation 3 calculates the expected delay of I_j if applying frequency boosting. Note that different from instance boosting, both the queuing and serving time reduce as the processing speed increases with higher frequency.

$$T_{freq} = \alpha_{lh} \times \left(\left(L_j - 1\right) \times \left(q_j + S_j\right) + s_j\right) \tag{3}$$

5.3 Adaptive Boosting

Based on the estimation of the expected delay with instance and frequency boosting, PowerChief adaptively chooses the boosting technique that reduces the expected delay of the bottleneck service most. Algorithm 1 gives the details for selecting the most beneficial boosting technique during runtime. PowerChief invokes *SELECTBOOSTING*(b_n) to reach a decision on which boosting technique to apply to the bottleneck instance b_n.

In Algorithm 1, the variable *avail* denotes the available power budget during runtime, whereas p denotes the power consumption of launching a new instance. It first tries to recycle power from running instances in order to acquire the power to launch a new instance exceeding the available power budget (line 7-10). After then, if the available power budget is still not enough to launch a new instance, the algorithm resorts to frequency boosting (line 11-12). To make a fair comparison, PowerChief first evaluates the power required to launch a new instance of the bottleneck service b_n. Then (line 15-24), it compares the expected delay of instance and frequency boosting, and chooses the one that results in the shortest expected delay. In addition, as shown in line 14 of Algorithm 1, if the realtime queue length of the bottleneck instance b_n is smaller than two, then launching a new instance hardly alleviates the load. In that case, PowerChief prefers frequency boosting to accelerate the bottleneck service instance. Note that the variables of r_1 and r_2 are execution times normalized to the service running at the slowest frequency. Since the frequency of r_2 is higher than r_1, the ratio of speedup regarding the boosted frequency T_f should be r_2/r_1.

Algorithm 1 Algorithm of adaptive boosting

```
Input: avail                                        ▷ Current available power budget
function SELECTBOOSTING(bn)
    p = bn.getPower()                               ▷ Current power of bn
    qt =bn.getQueuing()                             ▷ Average queuing time of bn
    st =bn.getServing()                             ▷ Average serving time of bn
    ql =bn.getQueueLength()                         ▷ Queuing length of bn
    if avail < p then
        rec = RECYCLE(p−avail)                      ▷ Algorithm 2
        avail += rec
    end if
    if avail < p then                               ▷ Cannot launch instance
        frequency boosting with avail power
    else
        if ql> 2 then                               ▷ Queue length larger than two
            T_i = (ql-1)*(qt+st)/2 + st             ▷ With inst. boosting
            r1 = bn.getSpeedup(bn.getFreq())
            freq = bn.calNewFreq(p)
            r2 = bn.getSpeedup(freq)
            T_f= r2/r1*((ql-1)*(qt+st)+st)          ▷ With freq. boosting
            if T_i < T_f then                       ▷ Expected delay
                instance boosting
            else
                frequency boosting
            end if
        else
            frequency boosting
        end if
    end if
    Update avail
end function
```

6 POWER REALLOCATION MECHANISM

To perform the boosting technique selected by the boosting decision engine, power reallocation across service instances is inevitable if the current power consumption already reaches the budget ceiling. However, to avoid generating new bottleneck services after power reallocation, it requires a careful design to find the right service instance to recycle the power as well as withdraw the service instance underutilized. In this section, we describe the power reallocation mechanism in PowerChief to address the above issue.

6.1 Power Recycling

Leveraging the results from the bottleneck identification process, it is straight forward to acquire a sorted service instance list based on the latency metric used to identify the bottleneck. Power recycling starts from the fastest service instance within the list that has less chance to generate a new bottleneck than the others. By taking advantage of the bottleneck identification process, it is easy to find the potential service instances to perform power recycling, without additional searching overhead. Note that the bottleneck identification process takes into account of queuing and serving delay as well as the runtime queue length, which implicitly reflects the stage dependency through the queuing behavior of each service instance.

Let I_0, ..., I_{k-1} represent the sorted service instances, where I_{k-1} is the service instance that has the largest latency metric (bottleneck service) and I_0 is the instance that has the smallest latency metric. If there is not enough power budget to perform the boosting technique to I_{k-1}, PowerChief recycles power allocation from I_0 first. If the available power budget is still not enough for the selected boosting technique even after the power budget allocated to I_0 is recycled to the minimum (frequency of I_0 is reduced to the lowest), PowerChief then recycles power allocation from the next fast instance (e.g., I_1). This procedure repeats until the available power budget is enough for boosting I_{k-1} with the selected boosting technique.

Algorithm 2 presents the persudo-code of recycling power allocation. If PowerChief decides to recycle power allocation of P (determined by boosting decision engine in Section 5.3), it invokes *RECYCLE(P)*. PowerChief employs greedy policy to recycle the needed power from the fastest service instances if possible. Other power recycling policies such as memory-bound instance first or maximum power saving per performance change can be easily plugged into PowerChief and replace current implementation. In general, we find the greedy policy performs well in practice.

Algorithm 2 Algorithm of power recycling

```
Input: I[k]                                     ▷ All the k instances (fast to slow)
Input: fl[k]                                    ▷ current frequency level of each instance
Input: p[b]                                     ▷ power at frequency level b (low to high)
function RECYCLEFROMINST(inst, power)
    cp = p[fl[inst]]                            ▷ Current power of inst
    recycled = 0                                ▷ Power recycled from inst
    for (int i = fl[inst]; i>=0; −−i) do
        recycled = p[fl[inst]] - p[fl[i]]
        if recycled >= power then
            break
        end if
    end for
    Scaling down frequency level of I[inst] to level i
    fl[inst]=i
    return recycled
end function
function RECYCLE(power)
    rec = 0                                     ▷ Already recycled power
    for (int i = 0; i < k; ++i) do
        rec+= RECYCLEFROMINST(i, power−rec)
        if rec >= power then
            break
        end if
    end for
    return rec
end function
```

6.2 Instance Withdraw

In addition to recycle power budget from an instance until it reaches the slowest frequency, PowerChief can also withdraw all the power budget allocated to a service instance by withdrawing it. Instance withdraw happens when PowerChief detects a service instance is underutilized. As mentioned in Section 4.1, PowerChief monitors

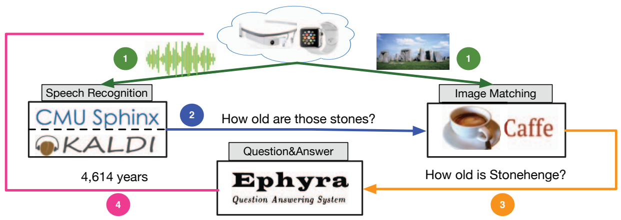

Figure 8: The service stages of Sirius application.

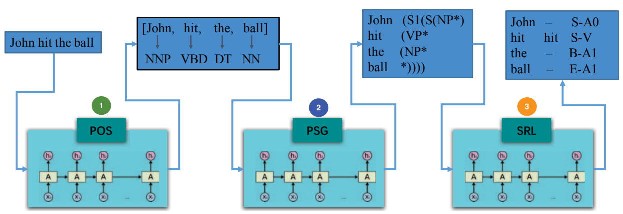

Figure 9: The service stages of NLP application.

the latency statistics of each service instance during runtime, it then calculates how much time each instance actually spends on processing queries during the withdraw interval. If the processing time is less than 20% of the withdraw interval, the service instance is considered underutilized and being withdrew to recycle the power budget.

To handle the load to the underutilized service instance before withdrawing it, PowerChief sorts the service instances within the same stage based on the latency metric used by bottleneck identification. The additional load is then redirected to the fastest service instance that has the least possibility to be overwhelmed. After assuring there is no query waiting or running on the underutilized service instance, PowerChief withdraws the service instance and recycles the power budget.

In order to avoid aggressive instance withdraw, at most one underutilized instance can be withdraw at each stage during one power reallocation interval. Also, an underutilized instance can be withdrew only if there are more than one instance within the same stage in case of breaking the application processing pipeline. Note that when a service instance is withdrew, its load migrates to existing instances. In such case, it generates negligible impact on existing instances since its load is already low (e.g., utilization is less than 20%).

7 REAL SYSTEM PROTOTYPE

7.1 Implementation Details

We implement a real system prototype of PowerChief to showcase its ability of intelligent power allocation for multi-Stage applications to improve responsiveness on power constrained CMP. For ease of adoption, we leverage the widely used RPC framework from Apache Thrift [3] to enable service stages interacting with each other seamlessly. In addition, services implemented in various programming languages can be easily hooked in the PowerChief through standardized APIs.

To demonstrate the advantage of PowerChief, we transform the Sirius and NLP (Senna [12]) into multi-stage applications by modularizing the processing pipelines into services. Both Sirius and NLP have three stages in their processing pipelines as shown in Figure 8 and 9. The Sirius application supports the user to submit queries with audio and image input, all the service stages process the queries in sequence to generate intelligent responses. The NLP application represents the semantic parsing of the text in natural language [18], which serves the automatic summarization commonly adopted in search engines.

We implement conventional boosting techniques such as frequency and instance boosting on top of the PowerChief runtime framework so that they can dynamically reallocate the power budget and boost the bottleneck service under the power constraint. The two boosting techniques are performed as follows: *1) frequency boosting* consistently increases the frequency of the service instance that is identified as bottleneck service; *2) instance boosting* always launches a new instance to accelerate the bottleneck service by sharing its load. The new instance takes the same frequency as the bottleneck service.

7.2 Overhead Analysis

The potential overhead that PowerChief may introduce can be inspected from three aspects, including latency monitoring, service startup/teardown and boosting decision. Latency monitoring timestamps the query when it is processed by a service instance (enter the queue, start processing and finish processing) to record the queuing and serving delay, which is negligible. To provide better user experience, modern service providers usually initialize sufficient service instances into a pool in advance [25]. Service startup/teardown actually means picking up a service instance from the pool (or returning back to the pool), which introduces negligible overhead. We adopt the similar idea to launch and withdraw service instance. The boosting decision may become a bottleneck when the number of services scales beyond a certain point. In that case, we can duplicate the services into multiple shardings [5] across CMP servers and use PowerChief to manage them separately with acceptable overhead.

8 EVALUATION

8.1 Experimental Setup

We evaluate PowerChief runtime framework on Intel Xeon E5-2630v3 server with two processors. Each processor has eight physical cores with SMT disabled. The processors use Haswell architecture, which supports DVFS on individual cores. The frequency can be adjusted from 1.2GHz to 2.4GHz with step of 0.1GHz. The operating system is Ubuntu 14.04 x86_64 with kernel 3.13.0-32. Since it is infeasible to measure power consumption at core level on current platform, we use the power model proposed in [22] to determine the power consumption of a core running at different frequencies. We treat the power consumption of the service instance as the power consumption of the core it is running on.

It is usually difficult to identify the optimal power allocation across stages due to the system dynamics (e.g., burst of load), which requires setting up the right number of service instances for each stage as well as right frequency for each service instance without violating the power budget. Thus we use stage-agnostic power allocation that divides the power budget equally across stages as our baseline. The power budget is chosen to accommodate one service instance running at the middle range of the frequency scale (1.8GHz) for each stage, so that PowerChief can exercise all its techniques throughout the experiments.

To evaluate the effectiveness of PowerChief under different load, we design a load generator that submits user queries following Poisson distribution that is widely used to mimic cloud workload [39, 41]. Three representative load levels (*high*, *medium* and *low*) are chosen throughout the experiments based on the extent how the service stages are saturated. To avoid the oscillation of power reallocation between the fastest and slowest services, we use a control variable *balance threshold*. When the difference of the latency metric between the fastest and slowest services is less than *balance threshold*, we skip the adjustment during current interval. The experiment configurations are summarized in Table 2. For the configurations of the applications, we use their default settings throughout our experiments. Note that although in this study we define stages based on application processing pipeline, our approach is also applicable to other stage definitions such as on the basis of algorithmic characteristics.

In Section 8.2 - 8.3, we evaluate the scenario where PowerChief is used to reduce the response latency while guarding the power budget. Whereas in Section 8.4, PowerChief is evaluated to reduce the power consumption while guarding the QoS. We compare the effectiveness of PowerChief and Pegasus in the latter scenario.

Table 2: Experiment setup of PowerChief in mitigating response latency under the power constraint. All services are running at medial frequency (1.8GHz)

Settings	Sirius & NLP
Load Distribution	Poisson
Load Level	High, Medium, Low
Stage Setup	1 ASR service, 1 IMM service and 1 QA service (Sirius); 1 POS service, 1 PSG service and 1 SRL service (NLP)
Power Budget	13.56watts
Adjust Interval	25 sec
Balance Threshold	1 sec
Withdraw Interval	150 sec

8.2 Intelligent Personal Assistant Application

In this section, we evaluate the effectiveness of PowerChief in mitigating response latency under the power constraint for Sirius application under different load.

Latency Improvement - Figure 10 shows the latency improvement for Sirius using PowerChief and other boosting techniques under different load. Compared to other boosting techniques, it is clear that PowerChief achieves the most latency reduction under all loads, with 20.3× (average latency) and 13.3× (99% tail latency) on average over the baseline. Especially under high load as shown in Figure 10(c), PowerChief significantly reduces the average and tail latency by 32.8× and 19.5× respectively over the baseline, which justifies the necessity for dynamically power reallocation and adaptive boosting.

We also notice that under medium and high load (Figure 10(b) and (c)), instance boosting performs better than frequency boosting, with average latency reduction by 24.46×(8.51×) and 25.11×(1.82×), whereas 99th percentile latency reduction by 14.46×(9.89×) and 14.77×(1.96×). However, as the load decreases the latency gap becomes smaller. At low load in Figure 10(a), frequency boosting reduces the average and tail latency more than instance boosting with 1.24× and 1.19× respectively.

This tendency is due to the fact that under medium and high load, the queuing time dominates the processing latency of the bottleneck service so that launching more instances alleviates the load and effectively reduces the queuing time. This observation is in accordance with previous work [28, 29]. Whereas under low load, the serving time takes a larger portion of the processing latency, therefore higher frequency is more beneficial in mitigating the bottleneck service.

The changing dominate factor for the processing latency at bottleneck service not only results from the load fluctuation, it is also affected by other runtime dynamics such as performance interference from collocated applications. The advantage of PowerChief in handling varying causes for bottleneck service with dynamic power reallocation and adaptive boosting maximizes the latency improvement under constrained power budget, thus is more promising in mitigating response latency for multi-stage application.

Effective Power Reallocation and Service Boosting - To illustrate the effect of dynamic power reallocation and adaptive service boosting, Figure 11 presents the runtime behaviors of the Sirius application under high load using frequency boosting, instance boosting and PowerChief. The same bottleneck identification method and power reallocation mechanism (without instance withdraw) from PowerChief is applied to frequency and instance boosting, except that PowerChief uses boosting decision engine to adaptively choose between the boosting techniques. Similar runtime behaviors are observed under medium and low load, and thus we omit them for brevity.

For frequency boosting in Figure 11(a), it first identifies the QA service instance as the bottleneck. Then it increases the frequency of QA service instance to 2.3GHz with power recycled from the IMM (frequency deceases to 1.2GHz) and ASR (frequency deceases to 1.6GHz) service instances during the first round of service boosting (at 25s). During the second service boosting interval (at 50s), the frequency boosting policy increases the frequency of ASR service instance to 2.1GHz by recycling the power from QA service instance (frequency decreases to 1.9GHz). When the load becomes low (between 175s and 275s), the serving time of QA service instance dominates the response latency, the frequency of QA service instance is boosted to its maximum(2.4GHz). In the rest of the query execution, power is assigned between QA and ARS service instances depending on which one is identified as the bottleneck service.

Different from frequency boosting, instance boosting launches a new instance when existing service instance is identified as bottleneck service. As shown in Figure 11(b), two more ASR service instances and three more QA service instances are launched with the power recycled through decreasing the frequency of existing service instances. It is seen that after 125s except one ASR instance (1.3GHz), the rest of the service instances all end up with the lowest frequency (1.2GHz). This is due to the fact that not enough power can be recycled to accommodate a new service instance even with the lowest frequency, which prevents instance boosting to perform further adjustment in response to varying bottleneck services.

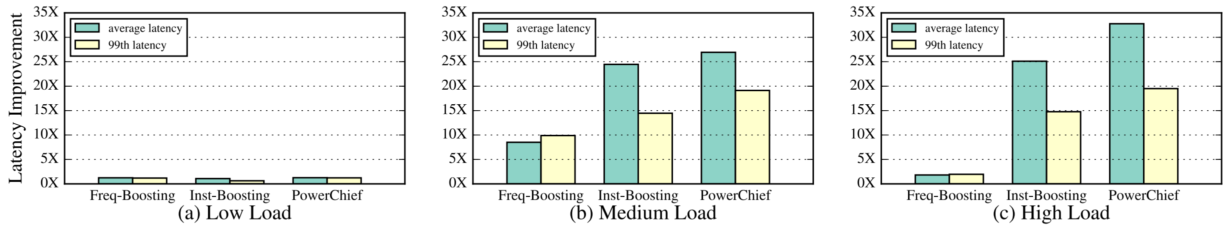

Figure 10: Latency improvement for Sirius application using PowerChief compared to other boosting techniques under different load. PowerChief achieves higher latency improvement under the same power budget.

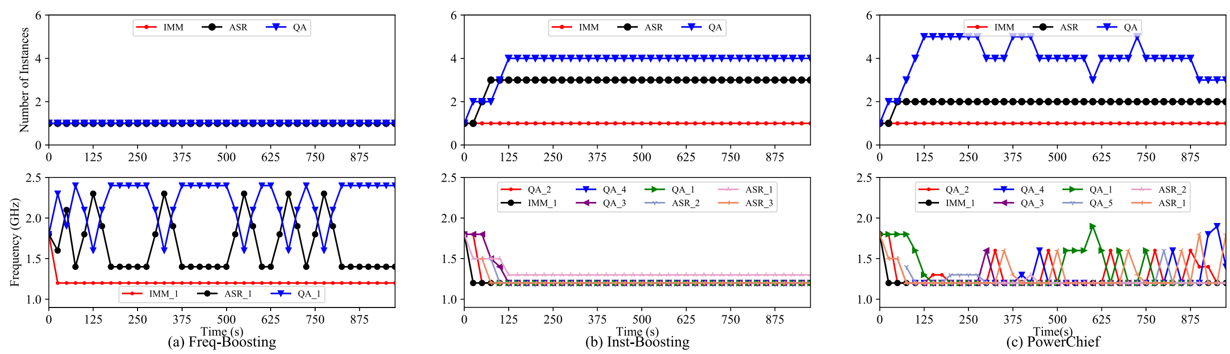

Figure 11: Runtime behavior of Sirius application such as the number and frequency of service instances across stages under different boosting techniques. Boost the bottleneck stage with (a) increased frequency, (b) more instances and (c) PowerChief.

To overcome the limitation of instance boosting that no power can be recycled if all instances are running at lowest frequency, PowerChief leverages the advantage of instance withdraw. This explains the number of QA instances drops from five to four after 300s in Figure 11(c), the power of which is utilized to boost the frequency of the bottleneck QA instance (QA_3, frequency increases to 1.6GHz). The boosting decision engine prefers instance boosting more over frequency boosting for QA service as the load increases (before 125s), since instance boosting absorbs the burst of load effectively to reduce the queuing delay.

The varying bottleneck services in Figure 11(c) is primarily due to the changing load during runtime. At the beginning QA service is identifies as the bottleneck. However, as more load arrives, the queuing delay becomes dominate at ASR service therefore it becomes the new bottleneck service. After boosting ASR, more load enters into the QA service, which becomes the bottleneck again. In addition, background activities (e.g., GC) as well as interference from collocated applications could also change the bottleneck service during runtime. These runtime uncertainties can be effectively handled by our approach.

8.3 Natural Language Processing Application

To demonstrate the ability of PowerChief in mitigating response latency under the power constraint within other application domains, we evaluate with the NLP application for semantic parsing of natural language.

Latency Improvement - Figure 12 presents the comparison between PowerChief and other boosting techniques in mitigating the average and 99% percentile latency for the NLP application under different load. Similar to the Sirius application, PowerChief achieves the most average and 99% latency reduction in all cases with 32.4× (average latency) and 19.4× (99% tail latency) on average over the baseline. Under high load as shown in Figure 12(c), PowerChief exhibits clear advantage in reducing average and tail latency by 52.2× and 28.4× respectively. At medium load in Figure 12(b), PowerChief shows similar average and tail latency improvement as instance boosting by 41.6× and 27.7×. Whereas at low load in Figure 12(c), PowerChief maintains similar average and tail latency improvement as frequency boosting by 3.4× and 2.3×. As the load decreases, the boosting decision adaptively prefers frequency boosting more from instance boosting, which effectively reduces serving time that dominates the latency at the bottleneck service.

8.4 Reducing power while meeting QoS

In addition to mitigate response latency under the power constraint, PowerChief is also capable to reduce power consumption of multi-stage application while meeting the latency QoS. The most related work in literature is Pegasus [34] that targets reducing power consumption without violating the QoS. In order to compare with Pegasus, we implement the Pegasus power conservation policy within PowerChief framework. The power conservation is the opposite of service boosting, which identifies the fastest service instance and applies frequency reduction and instance withdraw to save power

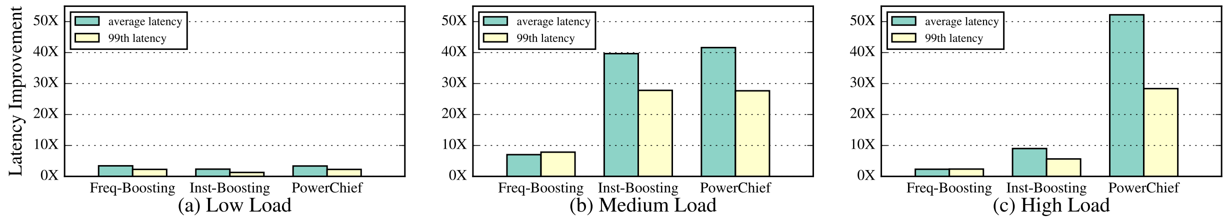

Figure 12: Latency improvement for NLP application using PowerChief compared to other boosting techniques under different load. PowerChief achieves higher latency improvement under the same power budget.

without violating the QoS. To make a fair comparison, we setup the frequency as well as the number of service instances within each stage so that the resource is over-provisioned regarding the QoS target. This is in accordance with the assumption of Pegasus. To demonstrate the ability in handling different stage organizations, we evaluate PowerChief with both Sirius and Web Search [2] applications. The detailed experiment setup is shown in Table 3. The results are normalized to the baseline where resource is over-provisioned and no power control is applied during runtime.

Table 3: Experiment setup to compare PowerChief and Pegasus in reducing power consumption while meeting the latency QoS. All services are running at maximum frequency (2.4GHz)

Settings	Web Search	Sirius
Adjust Interval	2s	10s
Stage Setup	1 aggregation service and 10 leaf services	4 ASR services, 2 IM services and 5 QA services
Power Conservation Policy	Frequency deboosting & Instance withdraw (PowerChief); Frequency deboosting (Pegasus)	
Latency QoS	250ms	2s

As shown in Figure 13 and 14, PowerChief conserves more power than Pegasus for both Sirius and Web Search applications while meeting the QoS target. For Sirius and Web Search, PowerChief saves 25% and 43% power over the baseline respectively, whereas Pegasus saves 2% and 10%. The fundamental advantage of PowerChief in conserving more power can be attributed to the acknowledgement of latency variation across service stages. During runtime, it identifies the fastest service instance across stages and adaptively applies power conserving policy without violating the QoS. Whereas, Pegasus treats service instances indifferently and thus cannot leverage the latency variations to trade QoS slacks for less power consumption.

8.5 Discussion

PowerChief manages dynamic power allocation at per application basis where each application has its own power budget and stage organization. It implicitly assumes each service instance is running on individual core where power management is applied. However, it is easy to extend the current approach to allow single service instance utilizing multiple cores. In the case of application collocation,

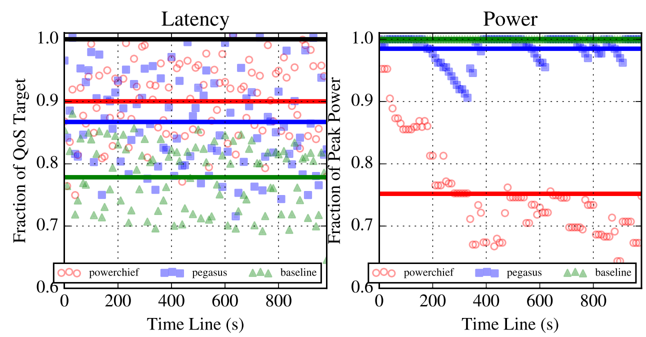

Figure 13: Power saving achieved by PowerChief and Pegasus with Sirius application while meeting the QoS target. Lines are average values across timeline.

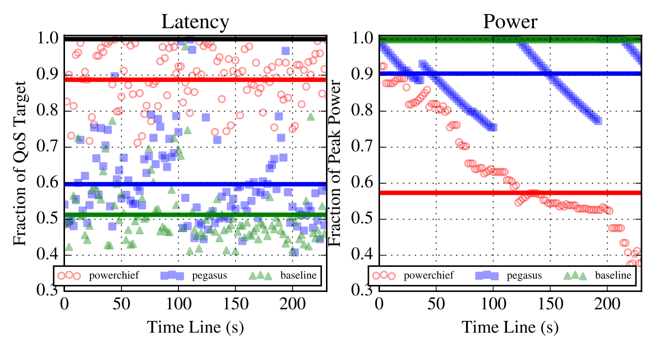

Figure 14: Power saving achieved by PowerChief and Pegasus with Web Search application while meeting the QoS target. Lines are average values across timeline.

as long as each service instance is running on physical cores exclusively, PowerChief is still capable to identify bottleneck service and perform power reallocation and service boosting. We do not consider to collocate multiple service instances on the same core since it could generate severe performance interference on shared resources and thus degrade the responsiveness. However, we admit even on separate cores, application collocation has the potential to generate performance interference and affect the effectiveness of our approach, which requires further investigation. Although in our current evaluation all application stages are running on a CMP server, there is

Table 4: Comparison between PowerChief and existing work from multiple aspects

	Pegasus [34]	Timetrader [58]	Kwiken [24]	Adrenaline [22]	Bubble-Flux [61]	Quasar [16]	PowerChief
Multi-Stage Awareness		✓	✓				✓
Power Constraint							✓
Commodity Hardware	✓		✓		✓	✓	✓
Runtime System	✓	✓			✓	✓	✓
Power Management	✓	✓		✓			✓

no constraint to run the stages in a distributed manner. Since all the components within PowerChief including the *CommandCenter* and *Stage* are implemented as services using Apache Thrift, they can communicate with the *CommandCenter* to enforce the power reallocation and service boosting decisions throughout the network. Note that the network delays are not considered in our study, however the joint design of service and query in our approach is extensible to include the network delays as measuring methods become available [1, 17, 42].

9 RELATED WORK

9.1 Guaranteeing the Response Latency

To address the response latency variation and provide guaranteed Quality-of-Service (QoS) [14, 33], research efforts are made from various aspects. Bubble-Up [38] quantitatively identifies resource interference under application collocation, and Bubble-Flux [61] dynamically manages the resource contention, to provide guaranteed QoS with increased utilization. SMiTe bounds the performance degradation on simultaneous multithreading (SMT) processors by carefully collocating "safe" applications through precise QoS prediction. Quasar [16] and Paragon [15] manages datacenter resources from multiple dimensions and use collaborative filtering to allocate the right type and amount of resources satisfying the QoS target. Compilation techniques [31, 55, 56] are proposed to guarantee the latency target by transforming code segments that cause severe performance interference. However, all techniques in this research category cannot be directly applied in power constrained scenario where our research proposal resides.

9.2 Improving Energy Efficiency

As the real hardware in datacenters is far from being energy proportional [6], research [27, 32, 39, 44, 48] is motivated to reduce power/energy consumption while guaranteeing the QoS target of user facing applications. Pegasus [34] insightfully identifies the latency slacks inside modern datacenters where resources are over-provisioned for peak load. Pegasus trades off the mean latency slacks for improved energy efficiency by slowing down the processing speed of the leaf nodes without violating the QoS target. Instead of average latency, Adrenaline [22] targets the tail of the latency distribution and only accelerates the queries that contribute to the latency tail through fine grained DVFS. TimeTrader [58] further extends the idea of trading off latency slacks for improved energy efficiency by exploiting the opportunity to slow down all leaf nodes that are not on the critical path to guarantee the latency target. However, these work implicitly assume that applications containing a single processing stage, and fail to acknowledge the intrinsic latency variance across multiple stages. This leads to diminished effectiveness in mitigating response latency for multi-stage applications under the power constraint.

9.3 Managing Latency of Multi-Stage Applications

Recent work [26, 57] confirms the benefits of application architecture that is composed of mulit-stage services for its flexibility and easiness of testability and deployment. Mitigating response latency across multiple processing stages is important for optimizing future cloud applications. However, research work in this direction are almost exclusive from giant companies such as Facebook [10] and Microsoft [24, 47] due to the lack of realistic multi-stage applications accessible to the academia. Contributed by the research effort from Sirius project [21], an open source multi-stage service based application is public accessible, which represents the emerging intelligent applications using the state-of-the-art implementation. This provides us a realistic application to study PowerChief on real system for mitigating response latency under the power constraint. There are some work [43, 54] propose adaptive parallelism to improve the performance as well as energy efficiency of pipelined application, however none of them deals with the QoS of user facing application under constrained power budget.

To sum up, Table 4 provides a comparison between PowerChief and existing work in terms of multi-stage awareness, power constraint, commodity hardware, runtime system and power management, which establishes the uniqueness of our work.

10 CONCLUSION

In this paper, we present PowerChief runtime framework that accurately identifies the bottleneck service and adaptively applies boosting techniques to mitigate the latency of multi-stage applications on power constrained CMP. Through evaluating our approach with Sirius and NLP applications, PowerChief improves the average latency by 20.3× and 32.4× respectively over the baseline, and 99% tail latency by 13.3× and 19.4×, while guarding the limited power budget. In addition, our QoS study demonstrates that PowerChief can also be applied to reduce power consumption while meeting the QoS of multi-stage applications. For both Sirius and Web Search applications, our approach saves 23% and 33% more power respectively than existing work regarding the same QoS target. For the future work, we are interested to apply our approach in production datacenter environment at large scale as well as analyze the tail latency behavior under the power constraint in more depth.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their feedback and suggestions. This work was partially sponsored by the National Key Research and Development Program of China (2016YFB1000503), the National Basic Research 973 Program of China under (2015CB352403), the National Natural Science Foundation of China (NSFC) (61502019, 61602301), the National Science Foundation (NSF) (IIS:1539011), NSF CAREER (SHF-1553485), IBM and Facebook.

REFERENCES

[1] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 19–19. http://dl.acm.org/citation.cfm?id=2228298.2228324

[2] Apache. 2010. Apache Nutch. (2010). http://nutch.apache.org/.

[3] Apache. 2010. Apache Thrift. (2010). https://thrift.apache.org/.

[4] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.

[5] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE micro* 23, 2 (2003), 22–28.

[6] Luiz André Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *Computer* 12 (2007), 33–37.

[7] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. 2006. Surf: Speeded up robust features. In *Computer Vision–ECCV 2006*. Springer, 404–417.

[8] J. Brutlag. 2009. Speed matters for Google web search. (2009). http://googleresearch.blogspot.com/2009/06/speed-matters.html.

[9] Edward A Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J Lambert, Kaladhar Radhakrishnan, and Michael J Hill. 2014. FIVRâĂŤFully integrated voltage regulators on 4th generation Intel® CoreâĎć SoCs. In *2014 IEEE Applied Power Electronics Conference and Exposition-APEC 2014*. IEEE, 432–439.

[10] MIchael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 217–231. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow

[11] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*. ACM, 160–167.

[12] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural Language Processing (Almost) from Scratch. *J. Mach. Learn. Res.* 12 (Nov. 2011), 2493–2537. http://dl.acm.org/citation.cfm?id=1953048.2078186

[13] J. R. Dabrowski and E. V. Munson. 2001. Is 100 Milliseconds Too Fast? (2001). In CHI.

[14] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[15] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 77–88.

[16] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 127–144.

[17] Mario Flajslik and Mendel Rosenblum. 2013. Network Interface Design for Low Latency Request-response Protocols. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 333–346. http://dl.acm.org/citation.cfm?id=2535461.2535502

[18] Daniel Gildea and Daniel Jurafsky. 2002. Automatic labeling of semantic roles. *Computational linguistics* 28, 3 (2002), 245–288.

[19] Google. 2015. Google RPC. (2015). http://www.grpc.io/.

[20] Sriram Govindan, Jeonghwan Choi, Bhuvan Urgaonkar, Anand Sivasubramaniam, and Andrea Baldini. 2009. Statistical Profiling-based Techniques for Effective Power Provisioning in Data Centers. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1519065.1519099

[21] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. 2015. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 13.

[22] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Lingjia Tang, Jason Mars, and Ron Dreslinski. 2015. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA) (HPCA '15)*. IEEE Computer Society, Washington, DC, USA, 10.

[23] David Huggins-Daines, Mohit Kumar, Arthur Chan, Alan W Black, Mosur Ravishankar, and Alex I Rudnicky. 2006. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, Vol. 1. IEEE, I–I.

[24] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding up distributed request-response workflows. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 219–230.

[25] Amin Jula, Elankovan Sundararajan, and Zalinda Othman. 2014. Cloud computing service composition: A systematic literature review. *Expert Systems with Applications* 41, 8 (2014), 3809–3824.

[26] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, David Brooks, and others. Profiling a warehouse-scale computer.

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

[28] 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 annual IEEE/ACM international symposium on Microarchitecture (MICRO-48)*.

[29] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads. In *Proceedings of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*.

[30] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. 2010. Server engineering insights for large-scale online services. *IEEE micro* 4 (2010), 8–19.

[31] Michael A. Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. 2014. Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (MICRO-47)*. ACM, New York, NY, USA.

[32] Jacob Leverich, Matteo Monchiero, Vanish Talwar, Partha Ranganathan, and Christos Kozyrakis. 2009. Power management of datacenter workloads using per-core power gating. *Computer Architecture Letters* 8, 2 (2009), 48–51.

[33] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. 2014. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[34] 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*. IEEE Press, 301–312.

[35] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 450–462.

[36] Jason Mars and Lingjia Tang. 2013. Whare-map: Heterogeneity in "Homogeneous" Warehouse-scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2485922.2485975

[37] Jason Mars, Lingjia Tang, and Robert Hundt. 2011. Heterogeneity in homogeneous warehouse-scale computers: A performance opportunity. *Computer Architecture Letters* 10, 2 (2011), 29–32.

[38] 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 (MICRO) (MICRO-44)*. ACM, New York, NY, USA, 248–259. https://doi.org/10.1145/2155620.2155650

[39] David Meisner, Brian T Gold, and Thomas F Wenisch. 2009. PowerNap: eliminating server idle power. In *ACM Sigplan Notices*, Vol. 44. ACM, 205–216.

[40] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. 2011. Power management of online data-intensive services. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 319–330.

[41] David Meisner and Thomas F. Wenisch. 2012. DreamWeaver: Architectural Support for Deep Sleep. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/2150976.2151009

[42] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2541940.2541965

[43] Jinsu Park and Woongki Baek. 2016. HAP: A Heterogeneity-Conscious Runtime System for Adaptive Pipeline Parallelism. In *European Conference on Parallel Processing*. Springer, 518–530.

[44] Steven Pelley, David Meisner, Pooya Zandevakili, Thomas F. Wenisch, and Jack Underwood. 2010. Power Routing: Dynamic Power Provisioning in the Data Center. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 231–242. https://doi.org/10.1145/1736020.1736047

[45] Nathaniel Pinckney, Matthew Fojtik, Bharan Giridhar, Dennis Sylvester, and David Blaauw. 2013. Shortstop: An on-chip fast supply boosting technique. In *VLSI Circuits (VLSIC), 2013 Symposium on*. IEEE, C290–C291.

[46] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. 2011. The Kaldi Speech Recognition Toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. IEEE Signal Processing Society.

[47] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[48] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. 2008. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1346281.1346289

[49] E. Schurman and J. Brutlag. 2009. The User and Business Impact of Server Delays, Additional Bytes, and Http Chunking in Web Search. (2009). http://velocityconf.com/velocity2009/public/schedule/detail/8523.

[50] Frank Seide, Gang Li, and Dong Yu. 2011. Conversational Speech Transcription Using Context-Dependent Deep Neural Networks. In *Interspeech*. 437–440. http://msr-waypoint.com/pubs/153169/CD-DNN-HMM-SWB-Interspeech2011-Pub.pdf

[51] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. 2013. Power Containers: An OS Facility for Fine-grained Power and Energy Management on Multicore Servers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2451116.2451124

[52] Kai Shen, Ming Zhong, Sandhya Dwarkadas, Chuanpeng Li, Christopher Stewart, and Xiao Zhang. 2008. Hardware Counter Driven On-the-fly Request Signatures. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/1346281.1346306

[53] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. *Google research* (2010).

[54] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. 2010. Feedback-directed Pipeline Parallelism. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/1854273.1854296

[55] Lingjia Tang, Jason Mars, and Mary Lou Soffa. 2012. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO) (CGO '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2259016.2259018

[56] Lingjia Tang, Jason Mars, Wei Wang, Tanima Dey, and Mary Lou Soffa. 2013. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS '13)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/2451116.2451126

[57] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2016. Workload characterization for microservices. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 1–10.

[58] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. 2015. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for On-line Search *(MICRO)*. ACM, Waikiki, Hawaii. http://arxiv.org/abs/1503.05338

[59] Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimpi, T Sivabalan, and Rajesh Subbiah. 2010. Worth their watts?-an empirical study of datacenter servers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 1–10.

[60] Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Chien-An Lai, Chien-An Cho, Yuji Nomura, and Calton Pu. 2014. Lightning in the Cloud: A Study of Very Short Bottlenecks on n-Tier Web Application Performance. In *Proceedings of USENIX Conference on Timely Results in Operating Systems*.

[61] 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 (ISCA) (ISCA '13)*. ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/2485922.2485974

[62] Yunqi Zhang, Michael Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS Prediction on Real System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (MICRO-47)*. ACM, New York, NY, USA.