TS-BatPro: Improving Energy Efficiency in Data Centers by Leveraging Temporal-spatial Batching

Fan Yao, Jingxin Wu *Member, IEEE,* Guru Venkataramani, *Senior Member, IEEE,* and Suresh Subramaniam, *Fellow, IEEE*

Abstract

With the rapid scaling of data centers, understanding their power characteristics and optimizing data center energy consumption is a critical task. Typically, data centers are provisioned for peak load even when they are mostly operating at low utilization levels. This results in wasteful energy consumption requiring smart energy saving strategies. At the same time, latency critical application workloads have stringent Quality of Service (QoS) constraints that need to be satisfied. Optimizing data center energy with QoS constraints is challenging since different workloads can have variabilities in job sizes and distinct system utilization levels. Also, server configuration (e.g., the number of cores per server) can be different across data centers. Therefore, a single configuration for energy management that works well across these various scenarios, is not practical.

In this paper, we propose *TS-BatPro*, a novel framework that judiciously integrates spatial and temporal job batching to save energy for multi-core data center servers while meeting the QoS constraints for application workloads. TS-BatPro performs effective global job batching and scheduling by modeling job performance and power characteristics of multi-core servers without any hardware modifications. TS-BatPro works on commodity server platforms and comprises two components: 1. a temporal batching engine that batches incoming jobs such that the servers can continue to reside in low-power states; 2. a spatial batching engine that prioritizes scheduling job batches to a small subset of servers. We develop a prototype of TS-BatPro on physical testbed with a cluster of servers and evaluate TS-BatPro on a variety of workloads. Our results show that TS-BatPro is able to achieve significant amount of energy savings under various job response time constraints and traffic patterns.

I. Introduction

Demands for personalized and contextual retrieval of large volumes of data from the users and the associated computations have strongly driven the growth of data centers. Today's computer systems are increasingly power hungry. Data centers now account for about 2% of the US domestic energy consumption [1], [2]. Most server farms are provisioned for peak demand, and configured to operate at capacities much higher than necessary. Studies by Barroso et al. [3]

have shown that the servers in data center environments are typically utilized at only 30% of their potential while drawing almost 60% of the power. The lack of server energy proportionality has significantly undermined data center energy efficiency and incurred considerable wasteful energy spent every year.

Two major issues contribute to the disproportionality between server utilization and energy consumption: *ineffective use of idle power modes in servers* that waste a considerable amount of energy by keeping excessive number of servers in standby mode (consumes 30% to 60% of server peak power) when they are idle; *over-provisioning of servers* which keeps CPUs in high power-consuming active state during the periods they are not processing any jobs.

Prior works that address data center energy issues can be broadly classified into three categories: (i) cluster-level power management techniques that dynamically size data centers by dispatching workloads to a subset of servers and put the rest of the servers in system low power states or just turn them off [4], [5], [6], [7]; (ii) server dynamic power management that leverage *Dynamic Voltage and Frequency Scaling* (DVFS) to achieve trade-offs between data center energy and application performance [8], [9], [10], [11]; (iii) recent works on server idle power management that take advantage of processor low power mode (e.g., *C State*) to conserve energy at low processor utilization levels while meeting the response time constraints [12], [13], [14].

While cluster-level energy optimization strategies can potentially save a large amount of energy by eliminating server platform power [14], they tend to be less effective for latency-critical workloads due to the long wakeup latencies (typically in minutes or tens of seconds). DVFS is shown to be effective in saving data center energy for ultra-short latency jobs with sub-millisecond service times [9], However, using DVFS, only CPU dynamic processor power can be addressed. Finally, energy savings could be obtained through smart control of core level low power states [12], [14]. However, merely achieving core level power saving can be sub-optimal. This is because, a significant amount of base power is drawn by the multi-core processor package when active [15]. To further improve energy efficiency, reducing the base processor power by putting the whole package to low power idle state is necessary. Unfortunately, entering processor-level idle state requires all of the cores to be in idle state at the same time, which is difficult to achieve as idle periods of individual cores rarely align [16].

In this article, we present TS-BatPro, an energy optimization framework that judiciously integrates temporal batching and spatial batching to improve data center energy efficiency. To

create chances for *processor-level low power states*, the temporal batching engine accumulates just the *right amount of jobs* before dispatching them to an individual server. To effectively bound the response latencies, our temporal batching engine uses a job performance model, that considers wakeup latencies from low power states and available parallelisms (number of cores in multi-core processor). Instead of simply balancing the workloads just by uniformly dispatching the workloads, our spatial batching engine maintains a *server status list*, and estimates the times when server will become idle. The spatial batching engine then dispatches the ready-to-execute job batch in a first-fit order to a server that is estimated to be idle. This further saves energy by packing the workloads onto a subset of processors.

TS-BatPro offers new technical contributions over our prior work [17] that leverages job batching to achieve processor-level energy savings in three major aspects: *First*, TS-BatPro uses a more rigorous statistical queuing model to estimate the right amount of tasks to batch for various system and workload configurations. *Second*, we implement a runtime load predictor module for system utilization that is used for determining the batching parameters dynamically. TS-BatPro adjusts its batching strategy to avoid deterioration of job tail latencies in cases of high load prediction errors. *Third*, TS-BatPro is evaluated on a variety of applications including workloads with real world traffic traces. TS-BatPro is shown to be adaptive under different workloads in term of job sizes and arrival patterns.

In summary, the contributions of our work are:

1) We highlight the necessity to understanding low power states and their power characteristics incorporated into modern multi-core processors in order to judiciously improve data center energy efficiency.

2) We propose and build TS-BatPro, a novel framework that performs temporal and spatial batching to optimize processor package-level sleep state residency that results in higher energy savings. We develop an effective analytical model that determines batching parameters with theoretical guarantees.

3) We implement a proof-of-concept system of TS-BatPro on a testbed with a cluster of servers and evaluate it with different workloads and utilizations levels, including real-world traffic traces. The results show that TS-BatPro is able to save significant amount of energy while still maintaining application QoS constraints.

II. Understanding Multi-core Processor Power Profile

To better understand the power and performance characteristics of modern server, we quickly review two critical concepts concerning multi-core processors. (i) A *core* is an independent processing unit that contains hardware execution contexts where the Operation System could schedule processes. (ii) A *processor*, also referred to as CPU, is a physical integrated circuit pacakge that may integrate multiple cores. Each core owns some private hardware components such as ALU and L1 cache. The processor also provides resources to be shared among all cores, i.e., last level cache and integrated memory controller.

Low-power State	Wake-up latency
Core sleep C1	1 μs
Core sleep C1-E	10 μs
Core sleep C3	59 μs
Package sleep C6	1 ms

TABLE I: Wakeup latencies for Core- and Package-level Sleep states on Intel Xeon processor [15], [18].

A. Processor Power Saving Mode

Emerging from embedded devices, low-power states are now an important feature targeted for power management in modern computer systems. The Advanced Configuration and Power Interface (ACPI) [19] provides a standardized specification for platform-independent power management. The ACPI-defined interfaces have been adopted by several major operating system vendors [19] and supported by various hardware vendors such as Intel and IBM [20], [21]. ACPI uses global states, *Gx*, to represent states of the entire system that are visible to the user. Within each *Gx* state, there are one or more system sleep states, denoted as *Sx*. For instance, *S0* is the working state and *S1* is the low-latency sleep state. Based on the ACPI specification, a processor core is allowed to be in a set of low-power states, i.e., *C* states, such as *C0*, *C1*, and *C6*. C0 is the active state, and the others are low-power states. A higher-numbered C state indicates more aggressive energy savings but also corresponds to longer wake-up latencies.

Modern processors generally provide high parallelism by integrating multiple cores within one package. Low-power C states are supported at both *core level* and *package level*. Core C state choices and residencies are generally determined by the Operating System (e.g., the *menu*

CPU-idle governor in Linux) based on applications' runtime activities. *The package* C *state is automatically resolved to the shallowest sleep state among all the cores*. Waking up from package C state takes longer time than the same level of core C state since the un-core components have to be activated before resuming the core execution contexts. Table I shows the wakeup latencies for various sleep states at core- and package-levels.

B. Multi-core Processor Power Profile

In order to effectively leverage processor low-power states to achieve energy savings, it is important to understand the power characteristics for multi-core processors under various core C state configurations. Figure 1 shows the power consumption for a 10-core Xeon E5-2680 processor with certain core C state enabled. We setup a micro benchmark to occupy a fixed number of cores n from 0 to 10. The idle cores are then allowed to enter a controlled C state, C_i. The processor power is read using Intel's Runing Average Power Limit (RAPL) interfaces [22]. From the figure we can see that the power proportionality towards number of active cores increases as deeper level C states are chosen.

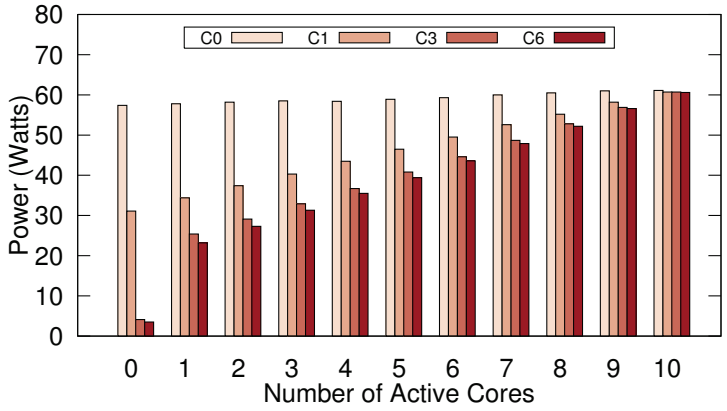


Fig. 1: Power range of a 10-core Xeon processor with different level of C state configurations.[1]

In the case when low-power C state is not used at all (C0), the power consumption is almost flat across different numbers of active cores. This indicates that even when all cores are idle, the processor consumes near-peak power, which is extremely energy inefficient. More importantly, for deeper C states, such as C3 and C6, besides power reduction with the decrease in the number

[1]We use a microbenchmark that can occupy a fixed number of cores with *taskset*. The rest of the idle cores are allowed to enter a controlled C state. Each power measurement is made using *RAPL* for a 5-minute run. Intel's Turbo Boost is disabled and the *performance* frequency governor is used to eliminate noise effect due to processor frequency fluctuations.

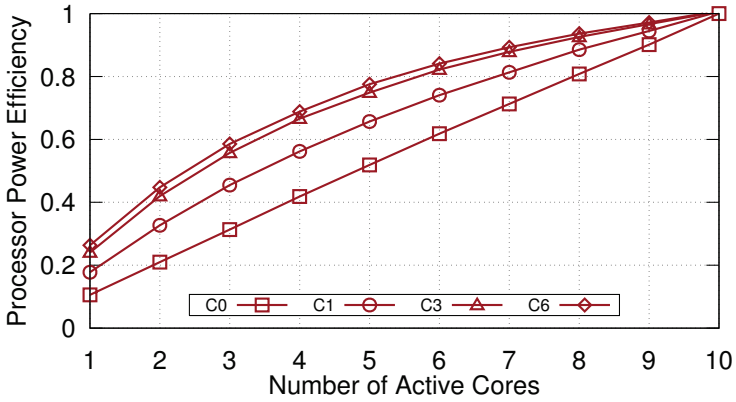


Fig. 2: Power efficiency of a 10-core Xeon processor with different level of C state configurations.

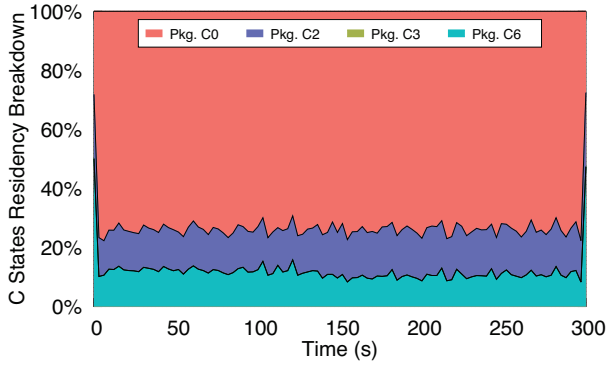
of active cores, there is a significant amount of power drop for the processor from having one core active to all cores idle. This is due to the fact that the processor package has to be in C0 (active) state whenever any of the cores are in C0 state. When all the cores are in low-power state C_i, the entire package can enter C_i state, which further saves power through power gating the resources that are shared by the cores. We define power efficiency for a multi-core processor as: $\frac{(P_{all-cores-active}/N)}{(P_{n-cores-active}/n)}$, where N and n represent the total number of cores and the actual number of active cores respectively.

Figure 2 shows the processor power efficiency with different numbers of active cores. We can see that the power efficiency increases with higher utilization. This indicates that, to save energy, two strategies need to be considered together:(i) increase the utilization of the cores in the multi-core processor so that it is operating in the most energy-efficient mode.(ii) keep all the cores idle so that a considerable amount of power could be saved using deep sleep state.

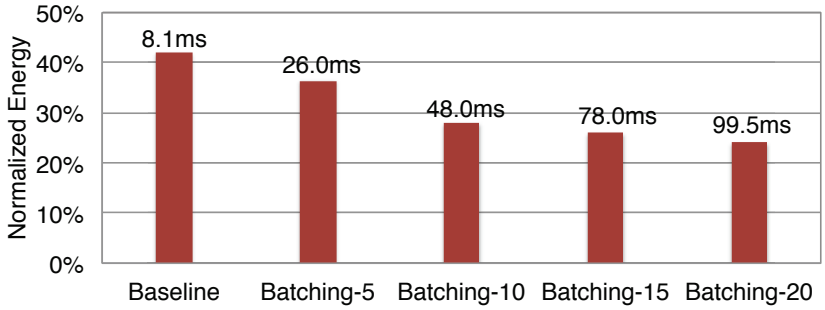
III. Motivational Example

As discussed earlier, modern multi-core processors consume a considerable amount of base power to keep the processor package active. Therefore, keeping the processor in package sleep state for a longer period of time is a straightforward strategy for saving processor energy, especially during periods when servers are underutilized. In order to reside in package-level low-power mode, all of the cores within the same processor need to be idle and enter the core C state first. However, due to the increasing core count in modern multi-core processors, the

[2]The C state residency is reported using *turbostat*. Due to limitation of the RAPL implementation on our platform, the *Package C0* represents the combined residence for package C0 and C1.



(a) C state residency breakdown-baseline.[2]

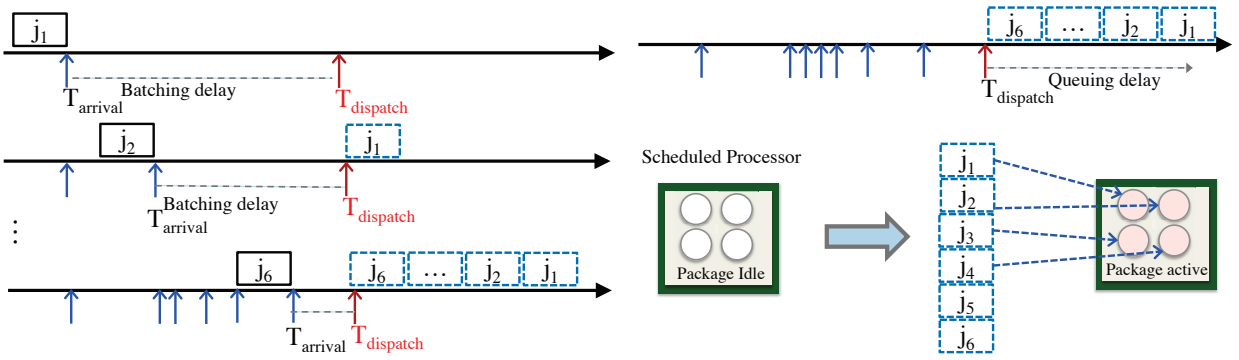


(b) Energy for baseline and simple batching mechanism.

Fig. 3: (a) shows the package C state residency breakdown for an Intel processor running a web server with an average 10% utilization; (b) illustrates the energy for baseline (without batching), Batching-5 and Batching-20 that accumulate 5, 10, 15 and 20 jobs respectively. The 95^{th} percentile latencies are shown above each bar.

busy and idle activities for individual cores could hardly synchronize without additional control at the processor level.

To demonstrate the package C state residencies, we setup a web application running Apache on the same Xeon-based server (also studied in Section II-B). The web application has an average service time of $5ms$. We use 95^{th} *percentile latency* for QoS analysis, which is common for latency critical workloads studies [23], [10]. We assume that the QoS constraint for the web application is $50ms$. Also, we consider a baseline algorithm that performs load balancing evenly across different cores and processors without explicit job batching. Figure 3a demonstrates the time spent in various package C states for the baseline algorithm under utilization of 10%. The plot shows that, even at the low utilization levels when the cores are supposed to be mostly idle, the processor spends very minimal time in the ultra power-saving package sleep (C6) state. To study the effect of batching, we develop a batching algorithm that simply batches a fixed number of web requests in the front end before dispatching the jobs to the server. Figure 3b shows the energy consumption for the baseline and two batching configurations that batches a



(a) Temporal Batching at Front End (b) Batched Jobs Processing at Worker Server

Fig. 4: An illustration of temporal job batching procedure assuming that the server is equipped with a 4-core processor. (a) shows that jobs are batched together before they are dispatched; (b) demonstrates that how the batched jobs are serviced at local server. Note that the first 4 jobs are processed simultaneously while the other jobs are queued.

range of jobs from 5 to 20.

The 95^{th} percentile latency is shown on top of each bar. Specifically, we could observe a tradeoff between energy reduction and the increase in tail latencies. For example, *Batching-5* achieves around 13% energy reduction compared to the baseline, and *Batching-20* yields almost 43% energy savings than the baseline. We note that with judicious batching, higher energy savings can be obtained at reasonable latencies: *conservative batching policies* only attain sub-optimal energy saving and leave considerable *latency slack* between the actual tail latency and the targeted tail latency (as in Batching-5); *aggressive batching policies*, though capable of saving substantial amount of energy, may significantly violate the QoS constraints due to the job queuing effect (as in Batching-20).

IV. System Design

In this section, we present the system design of TS-BatPro. TS-BatPro first performs temporal batching in the front end. Specifically, instead of dispatching job requests immediately to the individual servers, the temporal batching engine accumulates a certain amount of jobs and distributes the entire batch to a back end server. Essentially, this creates opportunities for the multi-core processors to use all of the cores at the same time (when the job batch arrives), thus improving the energy efficiency. As discussed in III, batching job requests aggressively can adversely impact the job response time. To maintain the Quality-of-Service for the jobs,

TS-BatPro integrates a two-stage queuing model that determines the maximum number of jobs to batch without violating the target latency constraints.

To further save energy, TS-BatPro incorporates a spatial batching engine that maintains estimated *to-be* idle time for each of the servers. TS-BatPro then schedules the job batch (from the temporal batching engine) to the first available server in a specific search order. Through spatial batching, jobs are concentrated on a small subset of servers such that the processors from the rest of the servers could stay in deep package sleep state without being unnecessarily woken up. The combined temporal and spatial batching make sure that significant processor energy could be saved while still maintaining the job QoS constraints.

A. Design of Temporal Batching

A large number of applications running on data center platforms (e.g., web service) are latency critical. The service providers for latency-critical applications will specify a target tail latency (e.g., 95^{th} percentile response time) as QoS guarantees. Typically, there is a latency slack between the application's average job service time and the target tail latency. As a result, we could take advantage of the latency slack by accumulating jobs such that processors can effectively utilize idle (low-power) states. We note that, as long as the tail latency constraints are satisfied, it is acceptable to delay executing the jobs. In our work, we assume a multi-server infrastructure where each server has parallelism due to the existence of multiple cores. We assume a FIFO job dispatching model where job requests arrive and get assigned in a first-in first-out order. Note that such queuing has been shown to be optimal for tail latency [24].

The challenging task of temporal batching is to determine the right number of jobs to batch based on the application workload and QoS requirements. In order to derive this batching parameter, we need to understand the various delays in the critical path of batched job processing. Figure 4 illustrates an example for such process. Specifically, Figure 4a shows the job batching at the front end. In this example, 6 jobs are batched before they are scheduled onto a server. For each job, it experiences a *batching delay* which starts from the time it arrives ($T_{arrival}$) to the time the entire batch gets dispatched. Figure 4b demonstrates the procedure for job processing on the local server side. Since the server have a 4-core processor, the first 4 jobs would be serviced concurrently while the rest two jobs will suffer from a *queuing delay*. Each core is working independently and will fetch a new job from the server's local queue once it finished its current job.

Determining the number of jobs in a batch. To derive batching parameter, we formalize the problem as the following: let K be the maximum number of jobs that would be batched temporally, $j_1, j_2, ..., j_K$ are the K jobs, the total delay for job j_i, D_i could be represented as:

$$D_i = B_i + N_i + \sigma. \tag{1}$$

where B_i and N_i are the batching delay and queuing delay for job j_i respectively and σ is a constant that represents the overhead of job dispatching, including overhead of batching operation and latency to wakeup a server that is currently in package sleep state.

Let $a_i, i = 1, \cdots, K$ denote the arrival time for each job. Then the batching delay for job i is defined as $B_i = a_K - a_i$. We use I_i to denote the interarrival time between job i and $i+1$, that is to say, $I_i = a_{i+1} - a_i$. Then the batching delay for job i is the sum of $K-i$ interarrival times:

$$B_i = \sum_{j=i}^{K-1} I_j. \tag{2}$$

Assume that the job arrivals to the system follow a Poisson distribution with arrival rate, λ. Then the interarrival times I between jobs are independent and identically distributed (i.i.d.) random variables with a common exponential distribution $F(x) = P(I \leq x) = 1 - e^{-\lambda x}, x \leq 0$ with mean $1/\lambda$. The sum of n i.i.d exponentially distributed random variables (r.v.s.) follows a $gamma(n, \lambda)$ distribution with the following probability density function:

$$f(x; n, \lambda) = \frac{\lambda^n \cdot x^{n-1}}{(n-1)!} e^{-\lambda x}. \tag{3}$$

Thus we can get the probability density function for B_i as a $gamma(K-i, \lambda)$ distribution, and the mean is:

$$E(B_i) = (K-i)/\lambda. \tag{4}$$

We use C to denote the total number of cores in a server. The K jobs will be dispatched to the cores. Assume that S is the job service time with an exponential distribution of rate μ. Obviously, the first C jobs can be directly assigned to the cores without waiting, thus we have $N_i = 0$, $i = 1, \cdots, C$. For job $C+1$, we have $N_{C+1} = \min(S_1, \cdots, S_C)$, which follows an exponential distribution with rate $C\mu$. As exponential distribution is memoryless, for job $C+2$, we have $N_{C+2} = N_{C+1} + \min(S_1, \cdots, S_C)$ which is the sum of two i.i.d. exponentially distributed r.v.s with a common rate $C\mu$. Thus N_{C+2} follows a $gamma(2, C\mu)$ distribution. Similarly, for job

$C+j$, N_{C+j} follows a $gamma(j, C\mu)$ distribution. The mean queueing delay for jobs i, $i > C$ is

$$E(N_i) = \frac{i-C}{C\mu}. \tag{5}$$

The total response time for a job i is $D_i + S_i$. To satisfy the QoS constraint, the 95^{th} percentile response time of a job should meet the target tail latency. Assume S^{95} is the 95^{th} percentile service time based on the exponential distribution with rate μ. Thus, for each i, we need to have:

$$B_i^{95} + N_i^{95} + \sigma + S^{95} \leq Q^{95}, \tag{6}$$

where Q^{95} is the target tail latency. As a result, K would be the maximum value that $\forall i \in [1:K]$, Equation 6 is satisfied.

Algorithm 1: Derive Param. K

Input: Service time distribution: S;
job arrival: λ, processor core count: C;
QoS: Q, batching overhead: σ;
Output: Batching param: K

let J_i be the i^{th} job in the batch;
let L_i be the tail latency of J_i;
derive batching delay distribution B_i for J_i;
derive queuing delay distribution N_i for J_i;
$R \leftarrow 0$;
$i \leftarrow 0$;
//check satisfiability of Equation 6
while $R < Q$ **do**
    i++;
    calculate tail latency L_i;
    R $\leftarrow L_i$;
K $\leftarrow i-1$;
return K;

Algorithm 2: Tmp. Bat. Runtime Control

Input: System utilization history: **U**;
load prediction window size: l;
predicted load at time t: P_t;
load prediction error threshold: thd;

let S' be the sampled service time distribution;
if $t < l$ **then**
    $K_t \leftarrow 1$;
    sample job service time and update S';
    use K_t for Tmp. Bat. in window t;
while $t \leq t_{end}$ **do**
    $P_t = AVG(U[t-1:t-l])$;
    get K_t with Algorithm 1 using P_t, S';
    //batching backoff
    if $|U_{t-1} - P_{t-1}| \geq thd$ **then**
        $K_t \leftarrow 1$;
    use K_t for Tmp. Bat. in window t;

The service time distribution could be monitored at runtime. Since the distribution typically does not change for a specific application, μ only needs to be profiled once (for example, in the warming up period of every workload). The value K could then be derived by repetitively incrementing K until Equation 6 is no longer satisfied. Algorithm 1 illustrates the routine of temporal batching that determines the batching parameter, K. We note that if the job arrival λ and the service time distribution S are known, K could be pre-computed. Particularly, it is

possible to compute various values of K for different QoS targets as a lookup table which can be looked up by the TS-BatPro runtime to avoid repetitive calculation. We also note that, when σ is sufficiently less than the job service time, the value of K can be independent of average job service time. Such observation can further be leveraged to reduce runtime computation overhead.

Our analytical framework models workloads with exponentially distributed service times and Poisson arrival process. Similar models have been used in several prior works [12], [25]. We observe that several realistic workloads evaluated in our work do not necessarily follow this distribution. In fact, these workloads have exhibited uniform distribution patterns. Therefore, our analytical model of Equation 6 would be too conservative for these workloads. Since it's too complicated to get closed-form system model with uniform distributed service times, we relax the constraints as the following:

$$E(B_i) + E(N_i) + \sigma + S^{95} \leq Q^{95}, \tag{7}$$

Section VII has demonstrated the validations of our accurate analytical model for workloads with exponentially distributed service times as well as the evaluation results for real workloads with the relaxed model.

Runtime control for temporal batching. When the system arrival rate is relatively stable at a certain level, TS-BatPro can take advantage of the same batching parameter for a long period of time according to Algorithm 1. However, real world traffic loads to data centers can exhibit fluctuations and even high burstiness. Therefore, using the same K is problematic as TS-BatPro may batch jobs too aggressively that results in violations of QoS, or too conservatively that ends with suboptimal energy savings. To overcome this issue, TS-BatPro integrates a runtime temporal batching controller that performs runtime system utilization prediction and batching adaptation. The algorithm for the runtime controller is listed in Algorithm 2. Specifically, the controller maintains the history of system loads. The load predictor computes the moving average with a length of l to predict next system utilization. Once the load is predicted, a new K value is computed and is used in the next epoch. Note that if the error of load prediction is high (e.g., in the presence of abrupt load changes), the batching parameter may not be accurate, which can lead to undesirable performance. TS-BatPro adjusts its batching strategy when it observes a high prediction error in the last sample. Specifically, the K value is reset back to 1 (i.e., no batching) when the error of prediction exceeds a threshold thd. TS-BatPro will later resume

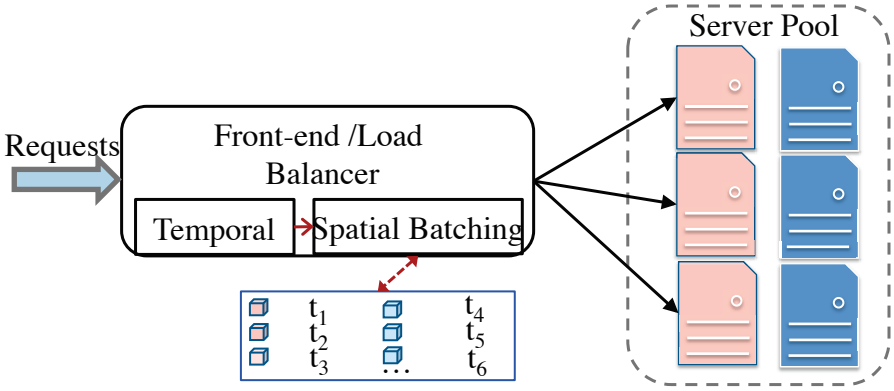


Fig. 5: Overview of Temporal and Spatial Batching mechanisms. t_i is the estimated processor idle time for server i. t_1, t_2 and $t_3 \geq t_{cur}$ (pink-colored servers), which means these servers are currently busy processing the batched jobs; t_4, t_5 and $t_6 \leq t_{cur}$ (blue-colored servers), indicating these three servers are idle.

batching when the load prediction error is low. Note that l and thd are also tunable parameters that can be adjusted based on the needs of the users and specific applications.

B. *Spatial Batching*

When a batch of jobs is generated by the temporal batching engine, the front end needs to find a server to process it. One possible way is to evenly distribute the loads to all the working servers. However, this approach is energy inefficient for the following reason: randomly dispatching job batches can create frequent active phases for all servers and not enough sleep periods. Since Operating System makes C state decisions based on server activity, it is possible that only shallow sleep state would be chosen because of insufficient opportunities for idleness. The spatial batching engine maintains a list that shows estimation times of when each of the servers would become idle. It then scans the list and find the first server which is supposed to be idle: $t_{current} \geq t_i$ for server i. It then updates the server's estimated idle time as $t_{current} + T_b$, where T_b is the estimated job batching time, which can be represented as $\lceil \frac{K}{C} \rceil * S^{95}$. Figure 5 shows the overview of our combined spatial and temporal batching approach.

V. Implementation

We implement a proof-of-concept prototype system including a load generator using httperf [26], TS-BatPro module and the apache HTTP servers on the back end. Httperf uses a open system model where the next job arrival is independent of the completion of the previous job. Prior

study has shown that load generators that utilize open system model can more accurately emulate real system traffic [27]. Httperf is modified so that it is able to generate loads to multiple apache servers. At the back end, the apache server is configured in the way that it always maintains exactly the same number of httpd processes as the number of cores. This makes sure that incoming batched jobs are processed in the queuing model as described in Section IV.

Our TS-BatPro is implemented as a separate module integrated into httperf. Once initialized, the temporal batching engine samples the services time and job arrivals to determine S^{95} and λ. After the two parameters are determined, it further derives K according to the methodology discussed in Section IV-A. The temporal batching engine then starts to perform job batching. Note that due to variations in job arrival rates, the temporal batching engine will setup a timer upon receiving the first job in each batch. The batching is complete either when K jobs are accumulated or when the timer expires, whichever comes first. This can avoid the cases where job arrival changes and the first job cannot wait until all K jobs arrive. λ is sampled every t seconds, which is a tunable parameter that controls response to load burstiness. The spatial batching engine chooses back-end servers based on its estimation of next server idle phase. Note that, to eliminate the potential of resource wear-out, the spatial batching engine would shuffle the order of the servers in the list every $T seconds$ so that all of them are exercised equally in the long run. We set t to 1 second and T to 60 seconds in our experiments.

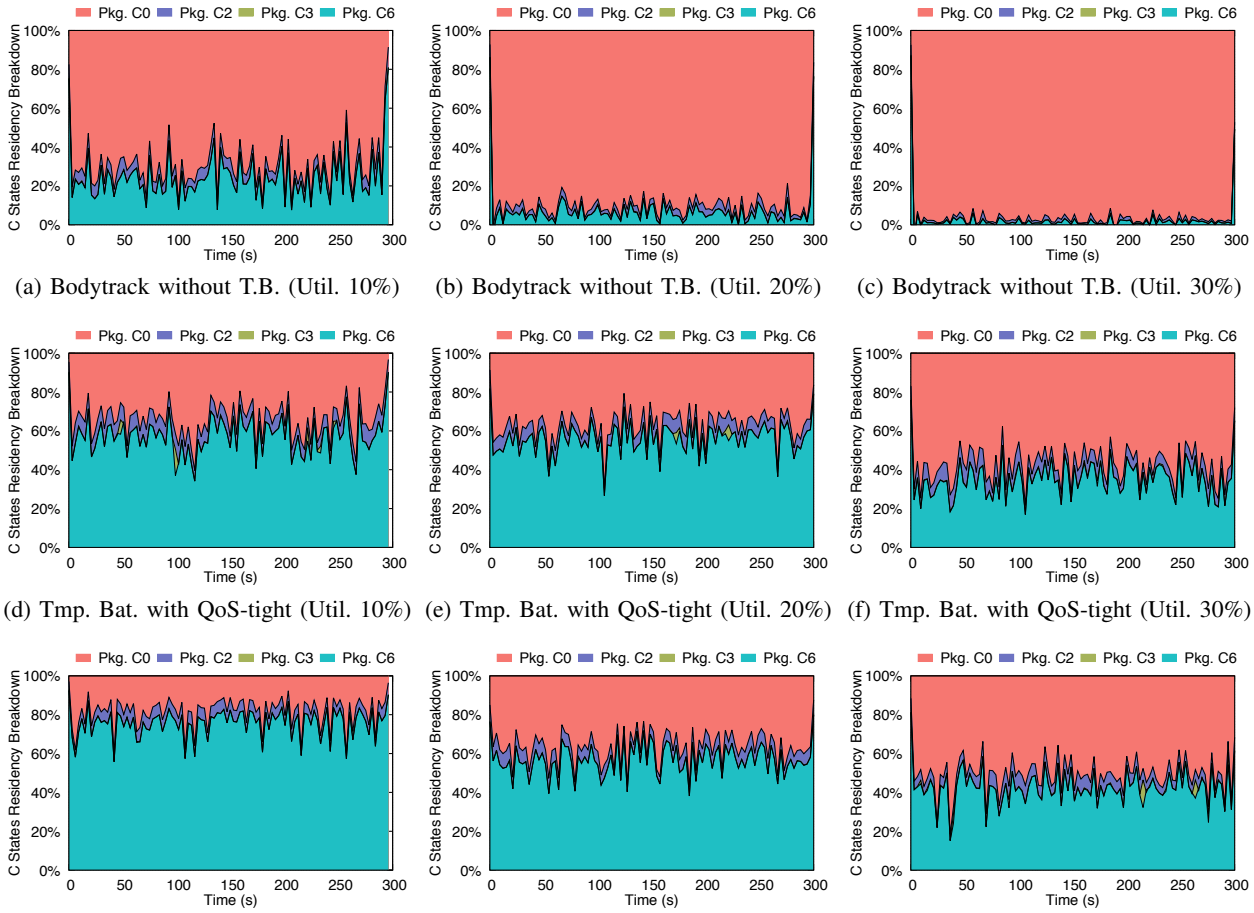
VI. Experimental Setup

Server platform. We deployed a testbed with a cluster of 18 servers. Two servers are working on the front end, within which one server generates job requests and the other server collects power measurements for the back end servers. In the back end, one Intel Xeon E5-2680 based server is used for fine-grained CPU power measurements (using the RAPL interface) in single-server experiment; the other 15 Intel Xeon E5650-based servers from the Dell Poweredge M1000e blade system are used for multi-server system evaluations. All of the back end servers are configured to run apache web service. These servers are interconnected with a NETGEAR 24-port Gigabit switch (star topology). We note that with this settings, all the network traffic is confined within the cluster, which eliminates unnecessary background traffic. Since the blade servers do not support RAPL interface, we utilize the IPMI interface for system-level power reading [28]. The server power consumption is queried and saved at every 1 second interval. We

conservatively set the end to end wakeup latency from package sleep state to 1 ms (the actual transition time is usually shorter than 1 ms [15]).

Benchmark selection and load generation. We run a set of benchmarks that cover a variety of latency-critical application characteristics in the cloud. We build a generic framework based on the apache web server. For each user request received, the apache application server runs the specific benchmark in the back end. We use the PARSEC [29] benchmarks as the back end service due to the fact that they are designed to represent latency-critical emerging workloads in the cloud computing environment including image processing, content search, and computer vision [30]. We develop CGI scripts for the Apache servers. The CGI script is flexible in that benchmarks could be easily integrated to run on the Apache framework. We select five applications: Bodytrack (108ms), Raytrace (79ms), Vips (42ms), Fluidanimate (33ms) and Ferret (21ms). The data in each brackets represents the average execution time for the application. Httperf is set to generate job arrivals based on exponential distribution. We configure httperf to generate three different levels of utilizations: 10%, 20%, and 30%. Additionally, we also use two real-world traces characterizing job arrivals patterns for *Wikipedia* [31] and *NLANR* [32], which exhibit different burstiness patterns. We create two synthetic benchmarks running in the backend for the two traces, with an average execution time of 5ms and 110ms respectively.

TS-BatPro parameter configuration. The batching parameter K is determined based on the algorithm shown in Section IV-A. In our experiment, we observe that the K value derived from the analytical model may violate the target QoS on a small number of occasions. One potential reason is that there exists resource contention between concurrent jobs. To sustain the QoS target, we set the actual value to $K-\varepsilon$. We observe that $\varepsilon = 2$ works practically well for all of our cases. Additionally, we have explored a variety combinations of l and thd, which denotes the size of load history and threshold for load prediction error respectively (See Section IV-A). Based on our observations, we set l to be 5 and thd to be 0.05. Finally, since the target QoS for different applications may differ, we define QoS as the tail latency normalized to the job's average service time. 95^{th} percentile response time is commonly set as the target SLA (service level agreements) for latency-critical workloads. We observe that most recent studies aim for a single tail latency for each benchmark [33], [23], [10]. We select two classes of tail latency targets in order to study the potential of TS-BatPro in energy optimization over both stringent and less-tight latency bounds. Typically the tail latency values range between 2.5x$\sim$10x. Correspondingly, we set 10x as the relaxed QoS constraint (QoS-relaxed) and 5x as the tight QoS constraint (QoS-tight) to



(a) Bodytrack without T.B. (Util. 10%) (b) Bodytrack without T.B. (Util. 20%) (c) Bodytrack without T.B. (Util. 30%)

(d) Tmp. Bat. with QoS-tight (Util. 10%) (e) Tmp. Bat. with QoS-tight (Util. 20%) (f) Tmp. Bat. with QoS-tight (Util. 30%)

(g) Tmp. Bat. with QoS-relaxed (Util. 10%)(h) Tmp. Bat. with QoS-relaxed (Util. 20%)(i) Tmp. Bat. with QoS-relaxed (Util. 30%)

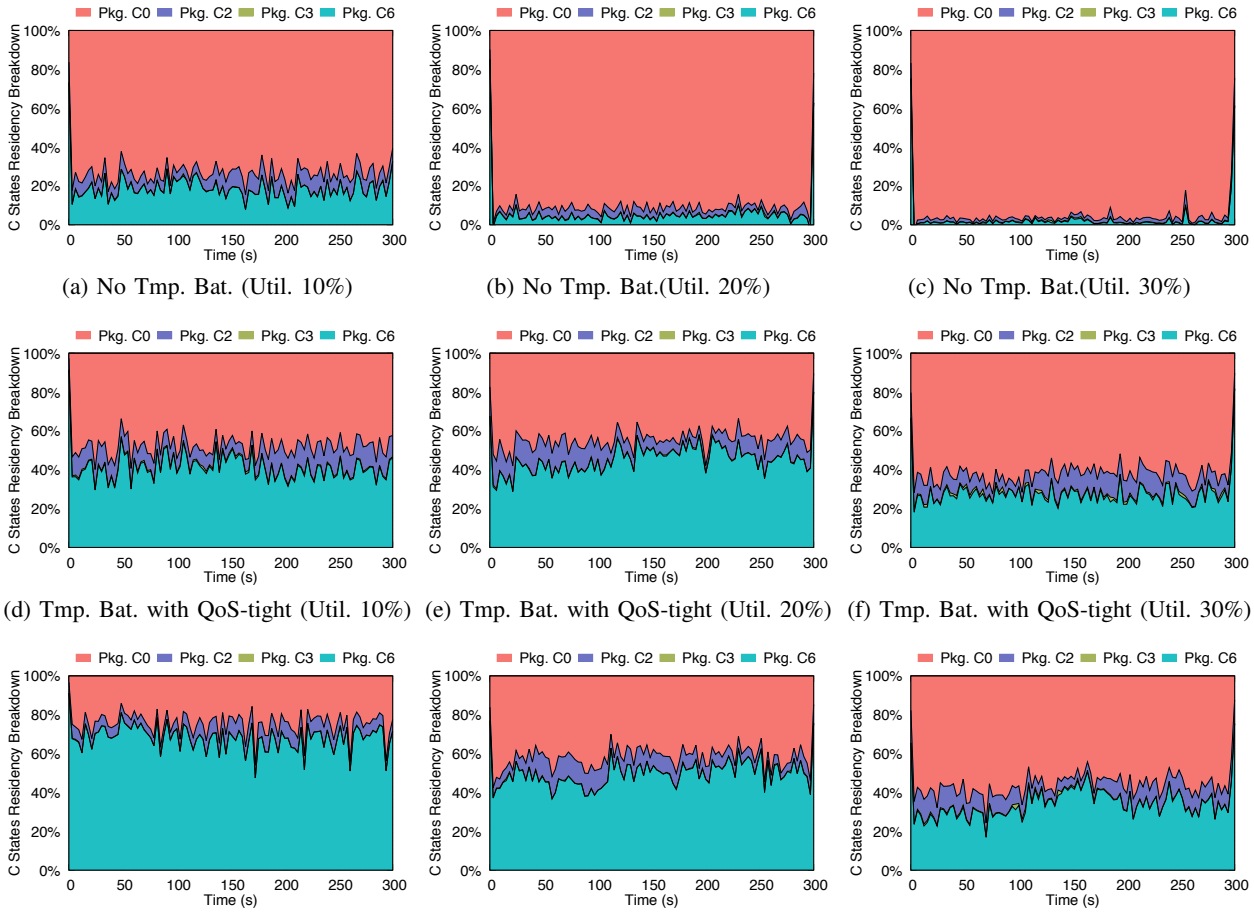
Fig. 6: Package C state residency breakdown for Bodytrack benchmark. Figure (a), (b) and (c) correspond to the residency breakdown with baseline configuration (no batching) under 10%, 20% and 30% system utilization respectively. Figure (d) (e) and (f) are for the same plots under Temporal Batching with tight Qos; Figure (g) (h) and (i) are for the same plots under Temporal Batching with relaxed QoS.

understand the implications of energy-latency tradeoffs with different user preferences for QoS.

VII. Evaluation of TS-BatPro

We evaluate TS-BatPro in two steps. Specifically, we first demonstrate the energy savings and job performance using just temporal batching on the Intel Xeon E5-2680 server. Then we enable both temporal and spatial batching engines on the blade system and illustrate the potential energy savings.

1) Temporal Batching Effectiveness: To evaluate the effect of temporal batching, we use a single Apache HTTP server. Httperf generates three different levels of system utilization



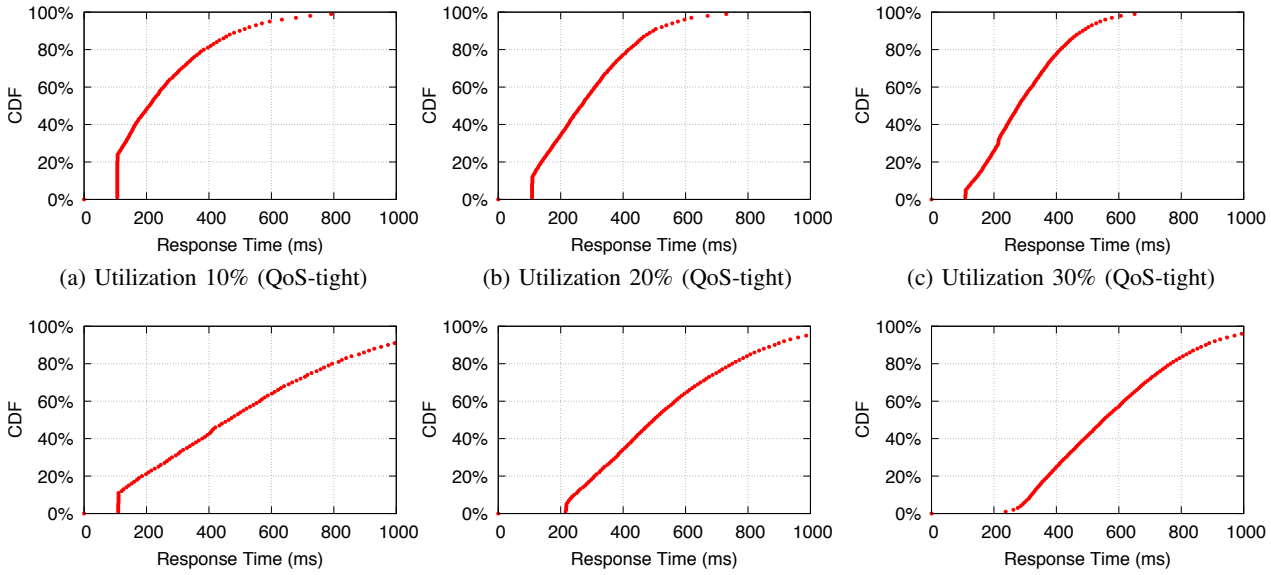
(a) No Tmp. Bat. (Util. 10%) (b) No Tmp. Bat.(Util. 20%) (c) No Tmp. Bat.(Util. 30%)

(d) Tmp. Bat. with QoS-tight (Util. 10%) (e) Tmp. Bat. with QoS-tight (Util. 20%) (f) Tmp. Bat. with QoS-tight (Util. 30%)

(g) Tmp. Bat. with QoS-relaxed (Util. 10%)(h) Tmp. Bat. with QoS-relaxed (Util. 20%)(i) Tmp. Bat. with QoS-relaxed (Util. 30%)

Fig. 7: Package C state residency breakdown for Vips benchmark. Figure (a), (b) and (c) correspond to the residency breakdown with baseline configuration (no batching) under 10%, 20% and 30% system utilization respectively. Figure (d) (e) and (f) are for the same plots under Temporal Batching with tight QoS; Figure (g) (h) and (i) are for the same plots under Temporal Batching with relaxed QoS.

levels:10%, 20% and 30%. For this experiment, we run the five PARSEC benchmarks. Note that the target tail latency (QoS) has to be provided to the temporal batching engine. The two QoS constraints are set in the temporal batching engine. For example, for Bodytrack, the two target latencies are 540*ms* (QoS-tight) and 1080*ms* (QoS-relaxed). Figure 6 shows the package C state residency for Bodytrack with and without temporal batching under the three system utilization levels using the two different QoS constraints. We can see that without batching, the processor spent less than 20% in the package C6 sleep state under 10% utilization (Figure 6a), which is significantly lower than the ideal residency of 90% under ideal energy proportionality. Server

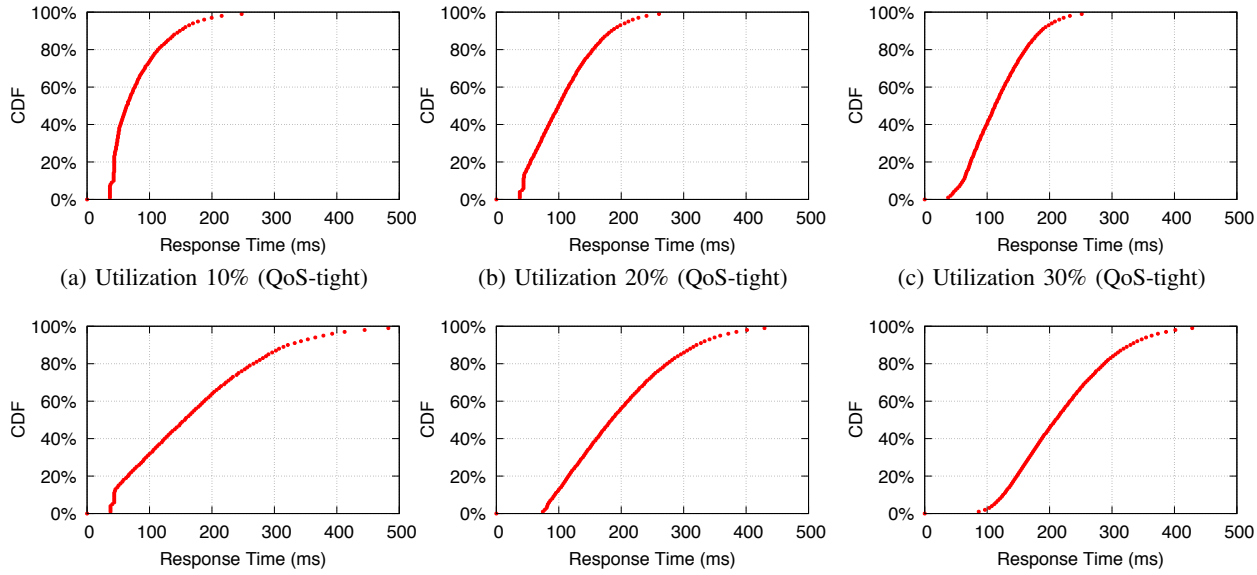


(a) Utilization 10% (QoS-tight) (b) Utilization 20% (QoS-tight) (c) Utilization 30% (QoS-tight)

(d) Utilization 10% (QoS-relaxed) (e) Utilization 20% (QoS-relaxed) (f) Utilization 30% (QoS-relaxed)

Fig. 8: Latency CDF for Bodytrack under 10%, 20% and 30% utilization using TS-BatPro's temporal batching.

residency in the power-saving states almost diminishes as the load increases to 20% and 30% (Figure 6b and Figure 6c). This clearly indicates the inefficiency of low power state management in under default OS settings. On the other hand, with temporal batching with TS-BatPro, the *Package C6* residency is significantly improved compared to the baseline without batching. For instance, the processor spent 41% more time in package C6 state under 10% utilization (Figure 6d), and spent 29% more time at 30% system utilization (Figure 6f). Meanwhile, we observe that the package C6 state residency increases as the target latency changes from QoS-tight to QoS-relaxed. This is because, since longer target latency allows for more aggressive batching, we observe higher chances of entering deep sleep state. Finally, the low-power state residency decreases much slower as the utilization level increases, compared to the baseline. Figure 7 illustrates the C state residency for the Vips benchmark. Similarly, the percentage of package C6 state residency is greatly increased under different system utilization levels. Moreover, we can see that, compared to Bodytrack, the C6 state residency is slightly less. We note that the job execution time of Vips is much shorter than that of Bodytrack. Therefore, under the same utilization, the inter-arrival times for *job batches* are relatively longer for Bodytrack, which favors entering of deep sleep state such as C6. Regardless, we can see considerable improvement in low power residency that will eventually reflect as energy savings.



(a) Utilization 10% (QoS-tight) (b) Utilization 20% (QoS-tight) (c) Utilization 30% (QoS-tight)

(d) Utilization 10% (QoS-relaxed) (e) Utilization 20% (QoS-relaxed) (f) Utilization 30% (QoS-relaxed)

Fig. 9: Latency CDF for Vips under 10%, 20% and 30% utilization using TS-BatPro's temporal batching.

Figure 8 and Figure 9 demonstrate the response time CDF for Bodytrack and Vips. We find that the temporal batching engine is able to meet the target constraint. For example, the actual tail latencies for Bodytrack (with average service time of 108ms) are 557ms and 986ms under 20% system utilization for QoS-tight and QoS-relaxed receptively. For Vips (with average service time of 42ms), the achieved tail latencies are 211ms and 358ms under 20% utilization using QoS-tight and QoS-relaxed. Notably, TS-BatPro can effectively shift the response time for various workloads regardless of the actual loads. We note that TS-BatPro's batching algorithm can successfully bound the target latency through batching for all the benchmarks.

Table II summarizes the energy savings of temporal batching for all five of PARSEC benchmarks in our study. Consistently, the energy saving increases as the QoS constraint is relaxed (e.g., from QoS-tight to QoS-relaxed) as we have observed before. As the utilization level increases, the energy saving reduces under all QoS settings in general. This is due to the fact to the processor idle intervals tend to be shortened with higher utilizations. It is also observed that batching can be more beneficial for applications with relatively larger job sizes (e.g. Bodytrack). Interestingly, when changing the system load from 10% to 20%, for each benchmark, TS-BatPro achieves more relative energy savings under QoS-tight than under QoS-relaxed. This is because under QoS-tight, the amount of batching is constrained by the target job latency especially at

	Utilization 10%		Utilization 20%		Utilization 30%	
	QoS-tight	QoS-relaxed	QoS-tight	QoS-relaxed	QoS-tight	QoS-relaxed
bodytrack (108ms)	31.9%	48.2%	33.4%	34.3%	20.3%	24.7%
fluidanimate (33ms)	13.5%	41.1%	18.2%	22.9%	8.7%	11.8%
vips (42ms)	21.0%	44.3%	27.6%	30.3%	16.0%	20.6%
ferret (21ms)	13.0%	41.9%	25.7%	32.2%	12.9%	22.6%
raytrace (79ms)	25.5%	46.7%	32.8%	34.8%	20.5%	26.4%

TABLE II: Power savings for all benchmarks using TS-BatPro's temporal batching. Energy savings are normalized to the baseline (OS default C state management) energy consumption.

	Utilization 10%		Utilization 20%		Utilization 30%	
	QoS-tight	QoS-relaxed	QoS-tight	QoS-relaxed	QoS-tight	QoS-relaxed
energy saving	10.2%	29.3%	7.1%	30%	6.6%	21.3%
performance	4.19x	7.55x	4.05x	7.33x	4.61x	8.35x

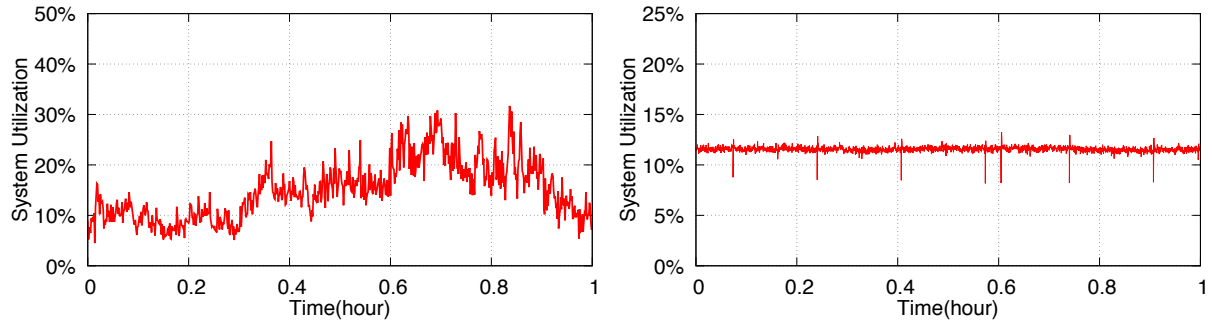
TABLE III: Power savings and performance using parameters derived from analytical model. Energy savings are normalized to the baseline energy consumption. Performances are normalized to mean service time.

lower utilization levels. Under QoS-relaxed, job batching is constrained by the higher job arrivals especially at the higher system utilization. Overall, we can see that temporal batching can save between 8.7% and 48% CPU energy depending on the workloads, server utilization levels and QoS constraints.

We also evaluate our analytical model in Equation 6 by applying the temporal batching to a synthetic workload whose service time is exponentially distributed with mean of 50ms. K values are obtained from Equation 6 and Algorithm 1 for different QoS and utilization levels. Table III summarizes the energy savings and performance of temporal batching for the synthetic workload. Consistently, the energy saving increases as the QoS constraint is relaxed (e.g., from QoS-tight to QoS-relaxed). We can also see that the performance constraints (in terms of 95^{th} percentile latency) were satisfied. Our analytical model finds appropriate K's to save energy without violating the QoS constraints.

A. TS-BatPro with Real World Traffic Traces

In the prior section, we have demonstrated the performance of temporal batching in TS-BatPro using a fixed job arrival rate with exponential inter-arrival time distribution. It is worth noting that in real world cases, system load is typically not known ahead of time. More importantly, the actual load may fluctuate over time. To evaluate the temporal batching performance of TS-BatPro, we build two workloads using two real world traffic traces. Each trace records the job



(a) NLANR bursty trace (b) Wikipedia non-bursty trace

Fig. 10: System utilization for two traffic traces for one hour: NLANR (Left) shows high level of burstiness while Wikipedia (Right) shows only marginal fluctuations.

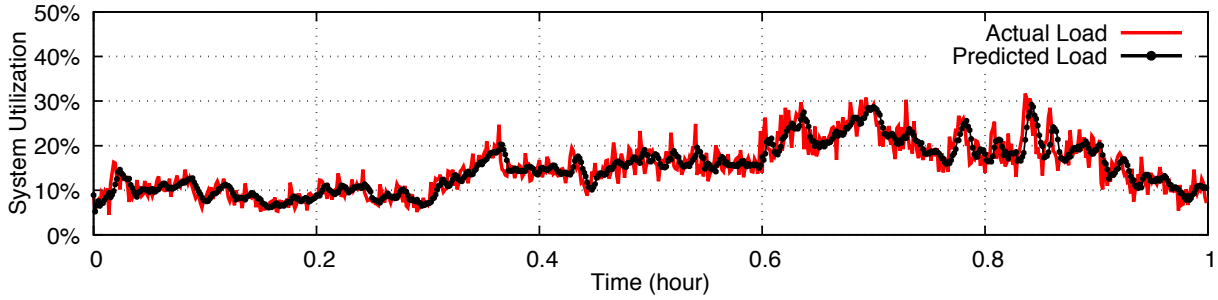
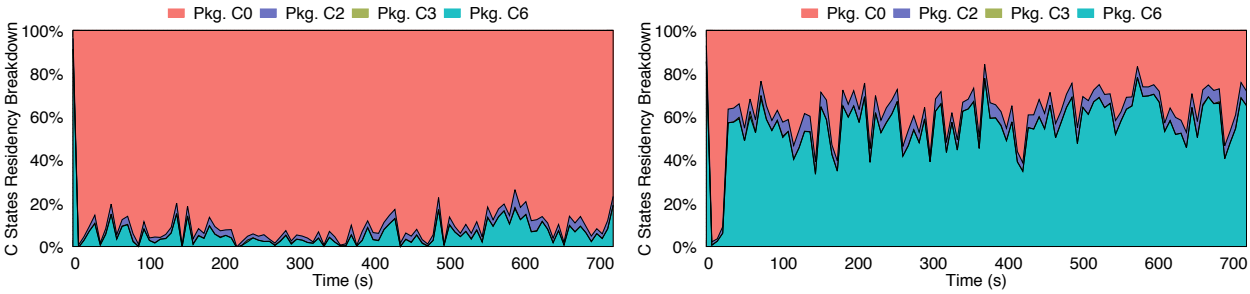


Fig. 11: TS-BatPro's predictions on NLANR traffic

request arrival timestamps that can be used to replay the network traffic. Figure 10 demonstrates the traces of utilizations for two workloads: the bursty NLANR workload [32] (Figure 10a), and the non-bursty Wikipedia workload [31] (Figure 10b) over one hour duration. As shown in our experiments, NLANR exhibits relatively high load fluctuations with a dynamic range of 29.0%. Differently, the Wikipedia trace only shows a few short spikes with most of the system utilization centered around 11.5%.

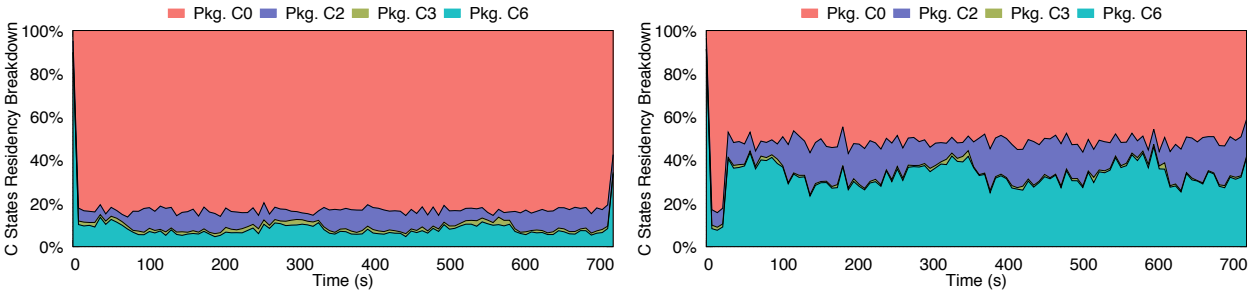
As discussed in Section IV, TS-BatPro incorporates a runtime load predictor to estimate the system utilization in the next time window in order to determine the batching size K. Figure 11 illustrate the original load and the predicted load over time. We can see with a moving average of 5 history samples, TS-BatPro is able to predict the system utilization with very high accuracy. Specifically, the average prediction error (i.e., difference between the actual loads and the predicated load levels) are less than 2% in terms of utilization. We note that TS-BatPro's predictions on workloads are performed at a much finer granularity (seconds), in comparison to peak/off-peak transition times (typically several minutes to hours). As a result, we observe very minimal changes of prediction error over long-period of traces. Additionally, inaccuracies in



(a) NLANR without Tmp. Bat.

(b) NLANR with Tmp. Bat.

Fig. 12: C-state residency for NLANR for the baseline and TS-BatPro's temporal batching



(a) Wiki without T.B.

(b) Wiki with T.B.

Fig. 13: C-state residency for Wikipedia for the baseline and TS-BatPro's temporal batching

load prediction would not introduce application performance violations as TS-BatPro cautiously disables batching when a high prediction error is observed.

We modified httperf so that the load generator can generate job arrivals by replaying the two traffic traces. We run the two trace-based workloads using the apache framework and set the target latency for both workloads to QoS-long. Figure 12 and Figure 13 illustrate the package C-state residency breakdown for the two workloads over a representative 12-minute interval. As expected, for NLANR, the baseline has very low C6 state residency (Figure 12a). TS-BatPro, on the other hand, creates over 60% C6 residency. Note that we can observe a sharp dip of C6 residency curve in Figure 12b and Figure 13b. This corresponds to the warming up period when the temporal batching engine monitors the job service times and collects samples of utilization history for further load prediction. Due to the burstiness of job arrivals in NLANR, we observe higher fluctuations of the C6 state residences as compared to Wikipedia. Moreover, as mentioned in Section IV, TS-BatPro dynamically halts job batching when the load prediction error is higher than a certain threshold (5% in our experiments). In our experiments, we see that only around 1.2% of time intervals in NLANR are subject to disabling of temporal batching (i.e., reset K

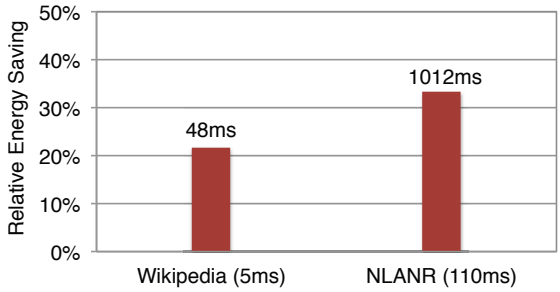


Fig. 14: Relative energy savings (with respect to the baseline without batching) and 95^{th} percentile latencies (shown above each bar) for Wikipedia and NLANR workloads with QoS-relaxed.

to 1) due to high load prediction errors while no K value reset is performed for Wikipedia. The energy savings and actual tail latency for these two workloads are shown in Figure 14. Overall, TS-BatPro has 22% energy saving on Wikipedia and 33% energy saving on NLANR while satisfying QoS constraints.

B. Combined Temporal and Spatial Batching

We perform both temporal and spatial batching on all the benchmarks as mentioned in Section VI at the system utilization level of 10%. The experiments are conducted on 15 Apache servers. The target tail latency is set to QoS-tight for all of the benchmarks. Figure 15 shows the overall energy savings for the entire cluster. Across all the benchmarks, temporal batching is able to achieve a steady energy savings between 48%-51%. TS-BatPro, with combined temporal and spatial approaches, can provide an additional upto 16% energy saving and achieves upto 68% saving compared to the baseline. We also observe that as the job size decreases, the relative energy saving increases. The reason is that with combined temporal and spatial approach, TS-BatPro is able to pack the loads to a fixed subset of processors, yielding similar system power among different benchmarks. Differently, in the baseline approach, shorted jobs tend to prevent processor from entering deep package sleep, significantly increasing the power consumption for the servers.

VIII. Discussion

Scalability of TS-BatPro. In the evaluation, we use a centralized controller for the temporal and spatial batching in TS-BatPro. This may cause some scalability issues when the data center

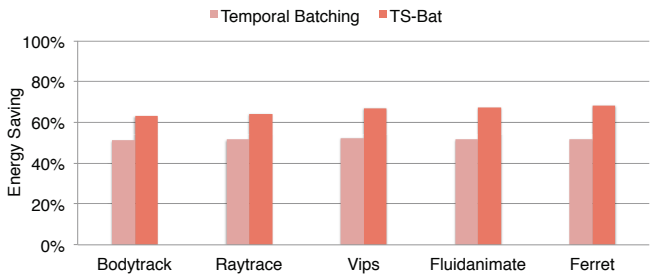


Fig. 15: Power savings for various benchmarks with Temporal Batching and TS-BatPro at 10% utilization. Energy saving is normalized based on the baseline energy.

has thousands of servers. We note that in these large scale data centers, TS-BatPro can be easily adapted with minimal modification. Specifically, we can divide the data center into multiple clusters, and each cluster will have its own TS-BatPro controller that coordinates the batching operation. Such scheme works well with many existing data center application as a lot of data center service are stateless [6]. That is, one user request can be serviced by any of the application servers in the pool. Eventually, each server cluster acts as a logical data center that can be effectively managed by TS-BatPro.

Energy optimization in data center networks. TS-BatPro largely considers energy optimization and QoS management for data center servers. We note that TS-BatPro can potentially benefit energy optimization in data center networks as well. Many prior works have proposed active power management on network devices using techniques such as dynamic link rate adaptation [34], [35]. However, merely reducing active power alone is not sufficient for network devices as a large portion of switch power is consumed simply by keeping the major components ON (e.g., line cards) [36]. Recent study has demonstrated the promise of using low-power states for both switches and servers to achieve high energy savings [37]. We note that TS-BatPro essentially proposes an effective scheduling framework that can be augmented to create idleness in both servers and network devices to achieve comprehensive energy savings in data centers.

IX. Related Work

Prior works [11], [8] have used DVFS based mechanisms to conserve server energy. Lo et al. [9] leverage *Running Average Power Limit* to dynamically adapt the runtime power of data center according to job latency feedback. However, at low server utilization, static power dominates and DVFS alone is not effective. Also, due to device scaling, the headroom for

voltage scaling has largely shrunk. As a result, techniques that address static power and energy are needed.

Maccio et al. [38] propose an analytical model to determine an optimal policy for on/off single server systems with FIFO allocation. Gebrehiwot et al. [39] study energy-performance tradeoff for a single server utilizing multiple sleep states. The server enters a random sleep state after some idling time, and restarts the server after a number of jobs are gathered in the queue. The system is modeled as a M/G/1 queue with Poisson job arrivals and general distributions of service time, setup delay and a timer. Sleepscale [12] jointly utilizes speed scaling and server sleep states to reduce the average power for single servers while satisfying the QoS constraint of normalized request latency.

To address energy-performance tradeoff in server farms, cluster level power management is used in prior works. Gandhi et al. [40], [6] develop mechanisms that reduce power consumption of the multi-server system by controlling the number of ON servers while satisfying the response time SLA. Yao et al. [41], [15] have demonstrated the adaptive use of system sleep states and/or CPU lower power state to tradeoff tail latency for increased energy efficiency. Differently, TS-BatPro is designed to optimize multi-core processor saving by leveraging low-power package sleep states. We note that these techniques are essentially complementary to TS-BatPro, and can be potentially integrated with TS-BatPro for more energy savings.

Knightshift [42] explores more specialized approaches such as exploiting heterogeneity of processor cores to improve energy. Two execution modes are utilized—one providing high performance while consuming higher power; the other being an active low power mode for low-utilization periods to save power. The model is extended by Wong et al. [43] to provide cluster-wide energy proportionality. However, to preserve generality of our solution and study the applicability of our techniques on many current warehouse scale systems, we model homogeneous servers and cores with same capability. We note that further power savings can be obtained at the application level through carefully tuning them for usage of processor resources [44], [45], load-balancing tasks across cores to avoid keeping cores unnecessarily active [46], and eliminating unnecessary cache misses that could potentially cut down power as well [47].

Tibor et al [4] studied the applicability of using multi-mode energy management in multi-tiered server clusters. A sleep state demotion algorithm is proposed based on analytical models. Several batching and scheduling mechanisms have been proposed for energy saving in server systems [48], [49]. Meisner et al. [16] propose architectural support to facilitate sleep state

management on multi-core servers that include scheduling policies to delay, preempt and execute requests and artificially create common idle and busy periods across cores of a server. However, the penalty of delaying and preempting requests are not considered. In our work, a realistic power model is used and implemented on real systems to evaluate the proposed policies. Finally, we note that server energy optimization proposed in TS-BatPro, can be integrated with more energy-efficient data center network topologies [50] to boost system energy savings.

X. Conclusion

In this paper, we propose *TS-BatPro*, an efficient data center energy optimization framework that judiciously integrates spatial and temporal job batching to save energy for multi-core data center servers while meeting the QoS constraints. TS-BatPro performs effective global job batching and scheduling by modeling job performance and power characteristics on multi-core servers. We developed a prototype of TS-BatPro on physical testbed with a cluster of servers and evaluate TS-BatPro on a variety of workloads. Our results show that the pure temporal batching achieves 49% percentage CPU energy saving compared to the baseline configuration without batching. Through combining temporal and spatial batching, TS-BatPro achieves upto 68% energy saving under various QoS constraints.

XI. Acknowledgments

This material is based upon work supported by the US National Science Foundation under grants CNS-1718133 and CAREER-1149557.

References

[1] R. Brown *et al.*, "Report to congress on server and data center energy efficiency: Public law 109-431," *Lawrence Berksleley National Laboratory*, 2008.

[2] J. Koomey, "Growth in data center electricity use 2005 to 2010," *A report by Analytical Press, completed at the request of The New York Times*, 2011.

[3] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," pp. 13–23, 2007.

[4] T. Horvath and K. Skadron, "Multi-mode energy management for multi-tier server clusters," in *PACT*.

[5] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu, "Delivering energy proportionality with non energy-proportional systems: Optimizing the ensemble," in *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, 2008.

[6] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems*, 2012.

[7] C. Isci, S. McIntosh, J. Kephart, R. Das, J. Hanson, S. Piper, R. Wolford, T. Brey, R. Kantner, A. Ng, *et al.*, "Agile, efficient virtualization power management with low-latency server power states," in *ACM SIGARCH Computer Architecture News*, vol. 41, ACM, 2013.

[8] D. C. Snowdon, S. Ruocco, and G. Heiser, "Power management and dynamic voltage scaling: Myths and facts," 2005.

[9] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *ISCA*, 2014.

[10] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 598–610, ACM, 2015.

[11] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *ISLPED*, 2007.

[12] Y. Liu, S. C. Draper, and N. S. Kim, "Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers," in *ISCA*, 2014.

[13] X. Zhan, R. Azimi, S. Kanev, D. Brooks, and S. Reda, "Carb: A c-state power management arbiter for latency-critical workloads," *IEEE Computer Architecture Letters*, 2016.

[14] C.-H. Chou, D. Wong, and L. N. Bhuyan, "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*, pp. 212–217, ACM, 2016.

[15] F. Yao, J. Wu, S. Subramaniam, and G. Venkataramani, "WASP: Workload adaptive energy-latency optimization in server farms using server low-power states," in *2017 IEEE International Conference on Cloud Computing (CLOUD)*, IEEE, 2017.

[16] D. Meisner and T. F. Wenisch, "DreamWeaver: architectural support for deep sleep," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 313–324, 2012.

[17] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "TS-Bat: Leveraging temporal-spatial batching for data center energy optimization," in *Global Communications Conference (GLOBECOM)*, IEEE, 2017.

[18] V. Pallipadi, S. Li, and A. Belay, "cpuidle: Do nothing, efficiently," in *Proceedings of the Linux Symposium*, vol. 2, pp. 119–125, 2007.

[19] M. P. T. HP, Intel, "Advanced configuration and power interface specification." http://www.acpi.info/.

[20] M. Floyd, M. Allen-Ware, K. Rajamani, B. Brock, C. Lefurgy, A. Drake, L. Pesantez, T. Gloekler, J. Tierno, P. Bose, and A. Buyuktosunoglu, "Introducing the adaptive energy management features of the power7 chip," in *IEEE Micro*, 2011.

[21] Intel, "Intel 64 and IA-32 architectures software developer's manual." http://www.intel.com/Assets/en_US/PDF/manual/253668.pdf.

[22] Intel, "Intel R 64 and IA-32 Architectures Software Developer Manual," *Volume 3b: System Programming Guide (Part 2)*, pp. 14–19, 2013.

[23] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, "Hipster: Hybrid task manager for latency-critical cloud workloads," in *2017 IEEE International Symposium on High Performance Computer Architecture*, pp. 409–420, 2017.

[24] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 9:1–9:14, ACM, 2014.

[25] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: Eliminating Server Idle Power," in *ASPLOS*, 2009.

[26] D. Mosberger and T. Jin, "httperf—a tool for measuring web server performance," *ACM SIGMETRICS Performance Evaluation Review*, 1998.

[27] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale," in *Proceedings of the 3rd Conference on Networked Systems Design & Implementation*, pp. 18–18, 2006.

[28] Dell, HP, Intel and others, "The Intelligent Platform Management Interface(IPMI)."

[29] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, ACM, 2008.

[30] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 33–47, ACM, 2016.

[31] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[32] "The nlanr project," 2012. http://www.nlanr.net.

[33] M. Alian, A. H. M. O. Abulila, L. Jindal, D. Kim, and N. S. Kim, "Ncap: Network-driven, packet context-aware power management for client-server architecture," in *2017 IEEE International Symposium on High Performance Computer Architecture*, pp. 25–36, 2017.

[34] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall, "Reducing network energy consumption via sleeping and rate-adaptation.," in *NSDI*, vol. 8, pp. 323–336, 2008.

[35] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu, "Energy proportional datacenter networks," in *ISCA*, vol. 38, pp. 338–347, ACM, 2010.

[36] T. Pan, T. Zhang, J. Shi, Y. Li, L. Jin, F. Li, J. Yang, B. Zhang, X. Yang, M. Zhang, H. Dai, and B. Liu *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1448–1461, 2016.

[37] B. Lu, S. S. Dayapule, F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "Popcorns: Power optimization using a cooperative network-server approach for data centers (invited paper)," in *2018 IEEE International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2018.

[38] V. J. Maccio and D. G. Down, "On optimal policies for energy-aware servers," *Performance Evaluation*, vol. 90, pp. 36–52, 2015.

[39] M. E. Gebrehiwot, S. A. Aalto, and P. Lassila, "Optimal sleep-state control of energy-aware m/g/1 queues," in *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '14, (ICST, Brussels, Belgium, Belgium), pp. 82–89, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.

[40] A. Gandhi and M. Harchol-Balter, "How data center size impacts the effectiveness of dynamic power management," in *Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2011.

[41] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A dual delay timer strategy for optimizing server farm energy," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 258–265, IEEE, 2015.

[42] D. Wong and M. Annavaram, "KnightShift: scaling the energy proportionality wall through server-level heterogeneity," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[43] D. Wong and M. Annavaram, "Implications of high energy proportional servers on cluster-wide energy proportionality," in *HPCA*, 2014.

[44] J. Chen, F. Yao, and G. Venkataramani, "Watts-inside: A hardware-software cooperative approach for multicore power debugging," in *International Conference on Computer Design*, pp. 335–342, 2013.

[45] J. Chen, G. Venkataramani, and G. Parmer, "The need for power debugging in the multi-core environment," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 57–60, 2012.

[46] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic, "Lime: a framework for debugging load imbalance in multi-threaded execution," in *International Conference on Software Engineering*, pp. 201–210, IEEE, 2011.

[47] G. Venkataramani, C. J. Hughes, S. Kumar, and M. Prvulovic, "DeFt: Design space exploration for on-the-fly detection of coherence misses," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 2, p. 8, 2011.

[48] S. Maroulis, N. Zacheilas, and V. Kalogeraki, "Express: Energy efficient scheduling of mixed stream and batch processing workloads," in *IEEE International Conference on Autonomic Computing*, pp. 27–32, IEEE, 2017.

[49] H. Goudarzi and M. Pedram, "Force-directed geographical load balancing and scheduling for batch jobs in distributed datacenters," in *IEEE International Conference on Cluster Computing*, pp. 1–8, IEEE, 2013.

[50] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A comparative analysis of data center network architectures," in *2014 IEEE International Conference on Communications (ICC)*, pp. 3106–3111, IEEE, 2014.