TS-Bat: Leveraging Temporal-Spatial Batching for Data Center Energy Optimization

Fan Yao, Jingxin Wu, Guru Venkataramani, Suresh Subramaniam
Department of Electrical and Computer Engineering
The George Washington University
Email: {albertyao, jingxinwu, guruv, suresh}@gwu.edu

***Abstract*—Data centers that run latency-critical workloads are typically provisioned for peak load even when they are operating at low levels of system utilization. Optimizing energy in data centers with Quality of Service (QoS) constraints is challenging since variabilities exist in job sizes, system utilization, and server configurations. Therefore, it is impractical to have a single configuration for energy management that works well across various scenarios.**

In this paper, we propose *TS-Bat*, a new data center energy optimization framework that judiciously integrates spatial and temporal job batching while meeting QoS constraints. TS-Bat works on commodity server platforms and comprises two major components: a temporal batching engine that batches the incoming jobs and creates opportunities for the processor to enter low power modes, and a spatial batching engine that schedules the batched jobs on to a server that is estimated to be idle. We implement a prototype of TS-Bat on a testbed with a cluster of servers, and evaluate TS-Bat on a variety of workloads. Our results show that pure temporal batching achieves 49% savings in CPU energy compared to a baseline configuration without batching. Through combining temporal and spatial batching, TS-Bat increases the energy savings by up to 68%.

I. Introduction

Today's computing systems are increasingly power-hungry. Data centers now account for about 2% of the US domestic energy consumption [1]. Most server farms are provisioned for peak demand, and configured to operate at capacities much higher than necessary. Studies by Barroso et al. [2] have shown that the servers in data center environments are typically utilized at only 30% of their potential while drawing almost 60% of the peak power. Lack of server energy proportionality has significantly undermined data center energy efficiency, resulting in wasteful energy spent every year.

Prior works on data center energy optimization can be broadly classified into three categories: (i) cluster-level power management techniques that dynamically re-size data centers by dispatching workloads to a subset of servers and turning off the rest of the servers or putting them into system (package) low-power states [3], [4], [5]; (ii) server-level dynamic power management that leverage *Dynamic Voltage and Frequency Scaling* (DVFS) to minimize energy while not adversely affecting application performance [6]; (iii) server-level idle power management that take advantage of CPU low-power modes (i.e., *C states*) to power-gate specific hardware structures and conserve energy [7], [8].

While cluster-level energy optimization strategies can potentially offer big energy savings through cutting down platform power [3], they tend to be less effective for latency-critical workloads due to the longer power-on (wakeup) latencies. DVFS is shown to be effective in saving energy for short latency jobs with sub-millisecond service times [6], but we note that DVFS is limited to active power only and is not effective for idle power. Finally, though a considerable amount of power could be saved through smart control of CPU idle states, merely optimizing core idle power can be sub-optimal during system runtime. This is because, for a multi-core processor, a significant amount of base power is drawn by the processor package to support shared hardware structures (e.g., L3 caches). We observe that more than 90% of power could be saved if the entire processor package is put into idle state instead of just individual CPU cores. Unfortunately, package level low-power mode requires all of the cores to be idle, which is difficult to achieve for multi-core processors since idle periods of individual cores rarely align [9].

In this paper, we present TS-Bat, an energy optimization framework that judiciously integrates a temporal batching engine and a spatial batching engine to save data center energy. To create opportunities 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, the temporal batching engine builds a job performance model based on the wakeup latency values from individual processor low-power states and the available amount of parallelism in the platform (i.e., number of cores per processor). The spatial batching engine then dispatches the ready-to-execute job batch to a server that is estimated to be currently idle. This further saves energy by packing the workloads on to just a subset of processors.

In summary, the contributions of our work are:

1) We motivate the need to develop data center energy saving mechanisms that are aware of multi-core processor performance and power characteristics in order to judiciously leverage their power-saving features (e.g., low-power states).

2) We propose TS-Bat, a novel QoS-aware data center energy optimization framework that combines temporal and spatial batching, and creates opportunities for CPU processors to enter highest energy-saving (package-level low-power) CPU modes both temporally and spatially.

3) We implement a proof-of-concept system of TS-Bat in a testbed with a cluster of servers, and evaluate the effectiveness

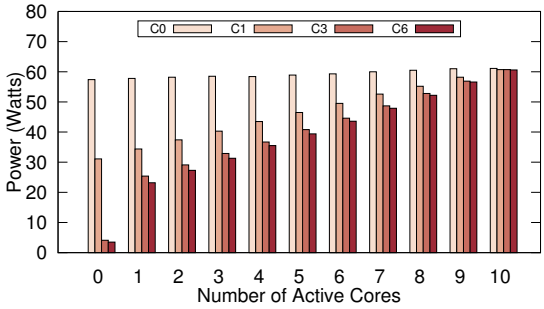


Fig. 1: Power range of a 10-core Xeon processor with different number of active cores and various C states configurations.[1]

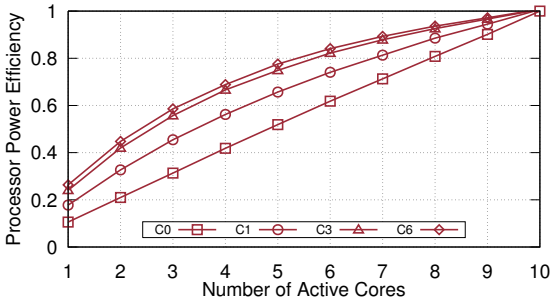


Fig. 2: Power efficiency of a 10-core Xeon processor with different level of C states configurations.

of our proposed framework on different types of workloads and various system utilization levels. Our experimental results show that TS-Bat is able to save significant amount of energy while maintaining the QoS constraints.

II. Background and Motivation

A. Processor Low Power States

The Advanced Configuration and Power Interface [10] provides a standardized specification for platform-independent power management. Based on the specification, a processor core can enter a series of sleep states (i.e., *C* states) such as *C0*, *C1*, *C3* and *C6*. In a C state, some architectural components are power-gated or put into low-power operating mode to conserve power. A higher-numbered C state indicates more aggressive energy savings at the expense of longer wake-up latency. Low-power C states are supported at both *core level* and *package level*. Core C state choices are generally determined by the operating system (e.g., the *menu* governor in Linux) when individual cores go idle. To enter a package C state where all the cores and the shared resources are set to low-power mode, all cores have to be idle first. The package *C* states are automatically resolved to the shallowest C state among the cores.

B. Processor Power with Low-power States

In order to effectively leverage processor low-power states, it is important to understand the power characteristics of multi-core processors under various C state configurations. Figure 1 shows the power profile for a 10-core Xeon E5-2680 processor when varying the number of active cores (idle cores are put to certain C state). The processor power is read using Intel's Running Average Power Limit (RAPL) interface [11]. We observe that the power proportionality increases as deeper level C states are chosen for idle cores. For deep sleep states, such as C3 and C6, we observe a significant power drop for the processor from having one core active to all cores idle (the first two groups of bars). This is due to the fact that the processor package has to be in C0 (active) state whenever any of the cores is active. When all the cores are in low-power state C_x, the entire package can enter C_x state, which further saves power through power-gating the shared resources such as the last level cache. Figure 2 shows the power efficiency for the processor with different numbers of active cores. The power efficiency is defined as: $(P_{all-cores-active}/N)/(P_{n-cores-active}/n)$, where n is the number of active cores and N is the total number of cores. We can see that the power efficiency increases as the processor is more utilized. This shows that two aspects need to be considered together: (i) increasing the utilization of cores in the multi-core processor so that it is operating in the most energy-efficient mode; (ii) keeping all the cores idle so that a considerable amount of power could be saved from the package-level, deep C state.

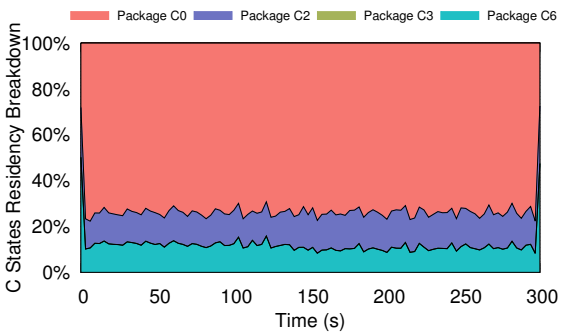
C. Need for Efficient Batching Techniques

As discussed earlier, to enter package low-power state, 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 busy and idle activities for individual cores rarely synchronize without additional control. As a result, the processor package can rarely go to sleep state naturally.

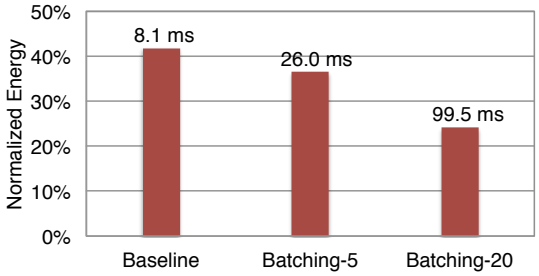
To study the package C state residencies, we set up Apache web server on a Xeon-based server (that we studied in Section II-B). The web application has an average request service time of $5ms$. Throughout this paper, we use 95^{th} *percentile latency* for QoS constraint. We assume that the QoS constraint for the web application is $50ms$. Also, we consider a baseline algorithm that performs load-balancing evenly across different cores. Figure 3a illustrates the fractions of time spent in various package C states over time for the baseline algorithm with under 10% system utilization. The plot shows that even at the low utilization level when the cores are supposed to be mostly idle, the processor spends a very small proportion of time in the C6 state.

We develop a simple batching algorithm that batches a fixed number of web requests in the front-end before dispatching them to the server. Figure 3b shows the normalized energy consumption for the baseline and two batching configurations that batches 5 and 20 jobs, respectively. The 95^{th} percentile

[1]We build a microbenchmark that occupies a fixed number of cores using *taskset*. The idle cores are set to enter a controlled C state. Intel's Turbo Boost is disabled and the *performance* frequency governor is used to eliminate noise effect due to processor frequency fluctuations.



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



(b) Normalized energy for baseline and simple batching mechanism.

Fig. 3: (a) Package C state residency breakdown for the processor running a web server with an average 10% utilization; (b) energy consumption for baseline (no batching), Batching-5, and Batching-20 that accumulate 5 and 20 jobs, respectively (normalized to energy consumption with C state disabled).

latency is shown on top of each bar. Our results show that even naive batching can save considerable energy. *Batching-5* achieves around 16% energy reduction compared to the baseline and *Batching-20* yields almost 43% energy savings. Clearly, we observe that batching should be judiciously used: *conservative batching policies* leave considerable *latency slack* from the target (seen in Batching-5); *aggressive batching policies*, though capable of saving substantial amount of energy, may significantly violate QoS constraints due to job queuing (seen in Batching-20).

III. System Design

In this section, we present the system design of TS-Bat. TS-Bat first performs temporal batching in the front end. Essentially, the temporal batching engine models the response time based on low-power state wakeup latencies, as well as server-side request parallelism. To maximize energy savings, TS-Bat incorporates a spatial batching engine that maintains estimated idle periods for each of the application servers. TS-Bat then schedules the job batch (from the temporal batching engine) to the first available server of an ordered list. Through spatial batching, jobs are scheduled onto a small subset of servers such that the remaining servers can continue to stay in deep package sleep mode.

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

A. Design of Temporal Batching

Data center service providers for latency-critical applications generally specify a target tail latency (e.g., 95^{th} percentile response time) for QoS guarantee. Typically, there is a latency slack between an application's actual job service time and the target tail latency. We can take advantage of this latency slack by accumulating jobs (temporal batching) before they are sent to the back-end servers.

The challenging task of temporal batching is to determine the right number of jobs (denoted as K) to batch so that the QoS will not be violated. In order to derive this parameter, we need to understand various delays in the critical path of job batching and processing. Figure 4 illustrates an example scenario. Specifically, Figure 4a shows the job batching at the front-end. In this example, 6 jobs are batched before they are scheduled to a server. Each job experiences a *batching delay* which starts from the time it arrives $T_{arrival}$ to the time when the entire batch gets scheduled on a server. Figure 4b illustrates the procedure for job processing at the local server. Since we have a 4-core processor, the first 4 jobs will be serviced simultaneously while the remaining two jobs will experience a *queuing delay*. To formalize the problem, let K be the maximum number of jobs that can be batched temporally, and j_i $(1 \leq i \leq K)$ is the i^{th} arrived job in the system. The total delay D_i for job j_i can be represented as:

$$D_i = B_i + U_i \tag{1}$$

where B_i and U_i are the expected batching delay and queuing delay for j_i, respectively. Assuming that S is the job service time distribution and λ is the job arrival rate for the system, we have the following:

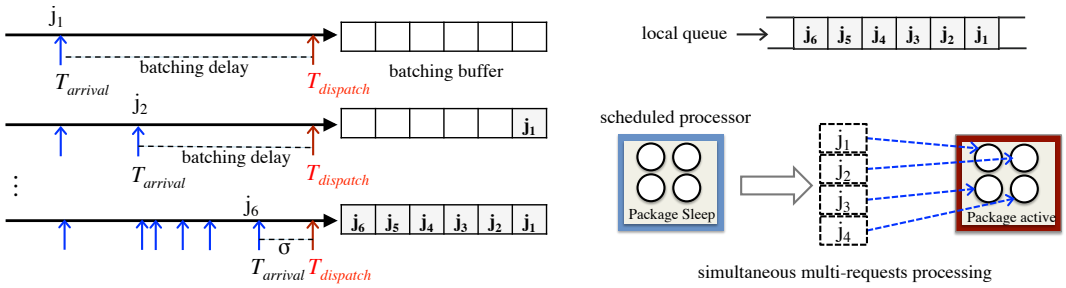
$$B_i = (K-i)/\lambda + \sigma \tag{2}$$

$$U_i = S^{95} * (i-1)/C + W \tag{3}$$

where σ is a constant that represents the overhead of batching, C is the number of cores per server, S^{95} is the 95^{th} percentile service time based on distribution S, and W is the wakeup latency for the processor in package sleep state. Based on these equations, K can be derived as the maximum value that satisfies the following inequality for all i:

$$D_i + S^{95} \leq Q \tag{4}$$

where Q is the target tail latency. The distribution S can be generated by monitoring the service times at runtime. We assume that the service time distribution does not change much over time, which is reasonable as data center operators typically do not mix latency-critical workloads with others [6]. As a result, S only needs to be profiled once (e.g., in the warm-up period of every workload). The value K can then be derived by repetitively incrementing K until inequality (4) is no longer satisfied. Since K is dependent on job arrival rate λ, the temporal batching engine periodically samples the job arrivals and updates the value of K. We note that once the distribution S is determined, the value K under different



(a) Temporal Batching in the Front-End (b) Batched Jobs Processing on Application Server

Fig. 4: An illustration of temporal job batching procedure assuming that the server is equipped with a 4-core processor. (a) shows how the jobs are batched together before they are dispatched; (b) illustrates how the batched jobs are serviced at the local server. Note that the first 4 jobs are processed simultaneously while the other jobs are queued.[3]

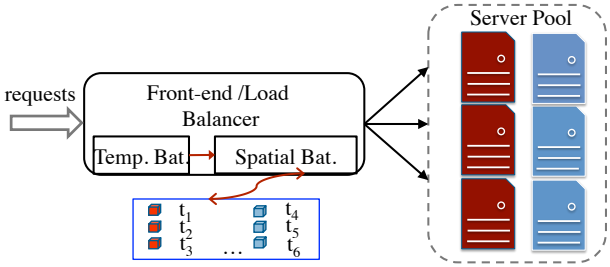


Fig. 5: Overview of overall TS-Bat scheme. t_i is the estimated processor idle time for server i. t_1, t_2 and $t_3 \geq t_{cur}$, which means these servers are currently busy processing the batched jobs; t_4, t_5 and $t_6 \leq t_{cur}$ indicating these three servers are idle.

arrival rates can be pre-computed. To avoid unnecessary delays due to having to batch K jobs (e.g., a sudden drop in arrival rate), an optional timer can be set to trigger dispatching of the currently accumulated jobs such that the earliest job will meet its deadline.

B. Spatial Batching

When a batch of jobs is accumulated 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 of the application servers. However, this approach is not energy-efficient because randomly dispatching job batches can create frequent active periods for all servers. Since the operating system makes sleep-state decisions based on server activities, evenly distributing workload may leverage only shallow sleep states in the absence of sufficiently long idle periods. To avoid this, we propose a spatial batching engine that determines the server to schedule the job batch. To do this, the spatial batching engine maintains a list that provides estimated times when the servers would become idle: $t_{current} \geq t_i$ where t_i is the estimated time when server i resumes idle and $t_{current}$ is the current dispatching time. 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 estimated as $\lceil \frac{K}{C} \rceil * S^{95}$. Figure 5 shows the overview of our TS-Bat approach.

[3]Data center latency-critical workloads (e.g., web server) utilize multi-threading mechanism in multi-core processors to improve overall throughput. Typically a local load balancer designates queued requests to multiple threads running in parallel.

IV. Implementation

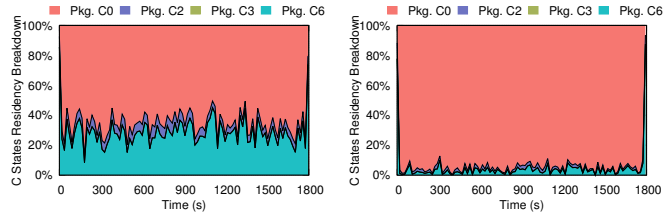
We implement a proof-of-concept prototype system including a load generator using httperf [12], a TS-Bat module, and apache HTTP servers in the back-end. httperf is modified so that it is able to generate loads to multiple apache servers. In the back-end, the apache server is configured in such a way 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 based on the queuing model described in Section III.

TS-Bat is implemented as a separate module integrated into httperf. Once initialized, the temporal batching engine samples the service times and job arrivals to determine S^{95} and λ. After the two parameters are determined, it derives K according to the methodology discussed in Section III-A. The temporal batching engine then starts to perform job batching. It sets up 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 is first. The batching buffer is set to 200 empirically, which is sufficient to hold requests for our workload with the smallest service time. The job arrival rate λ is sampled periodically every t seconds, where t is a tunable parameter that controls *TS-Bat*'s reactivity to load burstiness. By default, t is set to 5 seconds. The spatial batching engine chooses back-end servers based on its estimated idle period, and this information is stored in a linked-list.

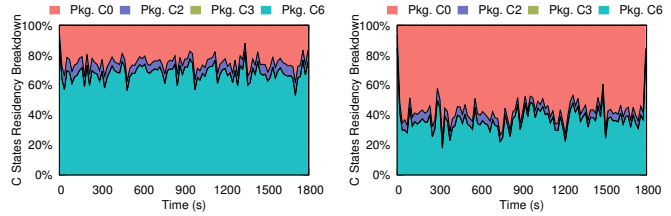
V. Evaluation

A. Experimental Setup

Server platform. We deployed a testbed with a cluster of 17 servers, including 2 standalone Xeon E5603-based servers and 15 Xeon E5650-based servers from the Dell Poweredge M1000e Blade system. The two Xeon E5603 servers are used as load generators and the blade servers are configured to run apache web service. All apache servers are interconnected with a Netgear 24-port Gigabit switch (star topology). Since our blade servers do not support RAPL interface, we utilize the IPMI interface for system-level power reading. Each Xeon

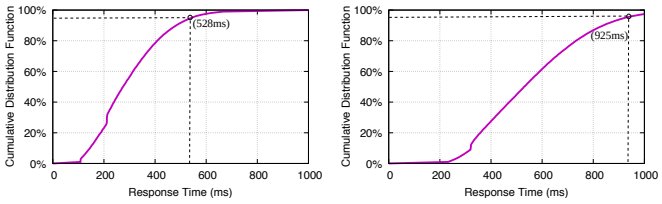


(a) Bodytrack without T.B.(Util. 10%) (b) Bodytrack without T.B.(Util. 30%)



(c) Bodytrack with T.B.(Util. 10%) (d) Bodytrack with T.B.(Util. 30%)

Fig. 6: Package C states residency breakdown for Bodytrack benchmark. (a) and (b) correspond to the residency breakdown with baseline configuration (no batching) under 10% and 30% utilizations respectively. (c) and (d) are for the same plots with Temporal Batching.



(a) Bodytrack (QoS 540*ms*). (b) Bodytrack (QoS 1080*ms*).

Fig. 7: Latency CDF for Bodytrack under 30% utilization using Temporal Batching.

E5650-based server is configured with the Apache HTTP server. The server power is queried and saved every 1 second. We conservatively set the wakeup latency from package sleep state to 1 ms (the actual transition time is usually shorter than 1 ms)

Benchmarks selection and load generation. To run varioius workloads, we developed CGI scripts for the Apache servers. We select a subset of the PARSEC [13] benchmarks to be executed by the CGI scripts. These PARSEC benchmarks represent emerging class of workloads from recognition, mining and synthesis domains of applications that can frequently benefit from running on the cloud. We generate workloads from five selected applications (with their average execution times shown in brackets next to them): Bodytrack (108ms), Raytrace (79ms), Vips (42ms), Fluidanimate (33ms) and Ferret (21ms). Each workload is configured to run for 30 minutes. 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%. Note that the target tail latency (QoS) should be provided to the temporal batching engine. We choose three levels of target tail latencies

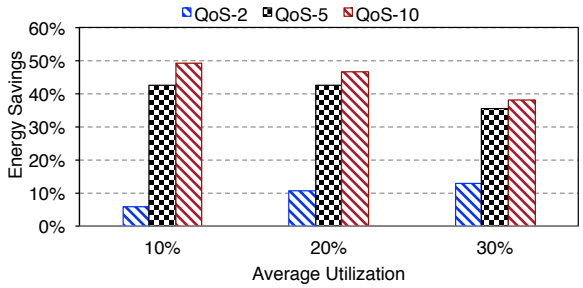


Fig. 8: Energy savings for Bodytrack using Temporal Batching compared to Baseline (no batching).

(normalized to average job service times), 2×, 5× and 10×.

B. Evaluation of TS-Bat

We evaluate TS-Bat in two steps. Specifically, we first show the energy savings and job performance using just temporal batching. Then we enable both temporal and spatial batching engines, and illustrate the overall energy savings that can be obtained from their combined deployment.

1) Temporal Batching Effectiveness: To evaluate the effect of temporal batching, we use a single Apache HTTP server. For this experiment, we use a single benchmark, Bodytrack. We observe that other benchmarks also exhibit similar result trends based on our experiments. For Bodytrack, the three target latencies are 215ms (QoS-2), 540ms (QoS-5) and 1080ms (QoS-10). Figure 6 shows the package C state residency for Bodytrack with and without Temporal Batching. We can see that the *Package C6* residency is significantly higher compared to the baseline (41% improvement under 10% system utilization levels, and 29% gain at 30% system utilization). This shows that TS-Bat can successfully create processor-level idleness that will translate to considerable amount of energy savings in the system. Figure 7 demonstrates the response time CDF when Bodytrack is set with QoS targets of 540ms and 1080ms. We see that temporal batching is able to meet the QoS target. Also, we observe that the achieved tail latencies are very close to the target latency, which indicates the accuracy of our performance modeling. Finally, Figure 8 summarizes the energy savings of temporal batching (normalized to the baseline without batching). As shown by our results, temporal batching can save upto 49% processor energy compared to the baseline without any job batching. As the levels of system utilization increase, the energy savings diminish for *QoS-5* and *QoS-10* settings. This is due to the fact that opportunities for processor idling tend to be smaller with higher levels of system utilization. Interestingly, *QoS-2* saves more energy with increasing system utilization. This is because latency slack is very small for this setting, and at low utilization levels, only a limited number of jobs can be batched.

2) Combined Temporal and Spatial Batching: We perform both temporal and spatial batching on all the benchmarks as mentioned in Section V-A at the utilization level of 30%. The experiment is conducted using 15 Apache servers. The target tail latency is set to 5× for all the benchmarks. Figure 9

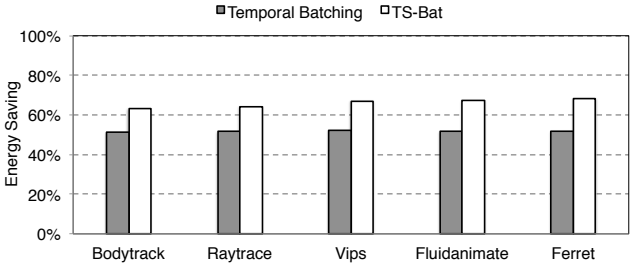


Fig. 9: Energy savings for various benchmarks with Temporal Batching and TS-Bat at 30% utilization. Baseline has no batching.

shows the overall energy savings for the entire cluster. Across all the benchmarks, temporal batching is able to achieve energy improvement between 48%-51%. TS-Bat, that combines temporal and spatial approaches, can provide up to 16% additional energy savings, and achieves up to 68% energy improvement compared to the baseline while meeting the target QoS constraints. We note that, with spatial batching, TS-Bat is able to pack the loads onto a small subset of processors. Differently, in the baseline approach, short latency jobs will incur more frequent arrivals, which prevents the processors from entering deep package sleep, thus significantly increasing the system power consumption.

VI. Related Work

Prior works have utilized DVFS-based mechanisms to improve data center energy efficiency by reducing the processor's active power. For example, Lo et al. [6] leverage *Running Average Power Limit* (RAPL) to dynamically adapt the operating frequency of data center servers based on feedback from actual job latencies. However, for systems with lower levels of utilization, static power dominates and DVFS alone is not effective.

Additionally, cluster-level power management mechanisms have been studied by several prior works [3], [4]. Autoscale [3] proposes a delayed-off mechanism that dynamically turns off servers after being idle for a preset period of time while satisfying the response time SLA. Due to the long wake-up latencies associated with the powered-off servers, these methods can incur unacceptable performance degradation for latency-critical workloads. WASP [8] takes advantage of both processor and system low-power states to optimize energy efficiency; however it does not consider batching to create opportunities for package-level idleness. With explicit package-level sleep control, TS-Bat can achieve significant amount of processor energy savings.

Prior works have adopted temporal batching to optimize system power. Dreamweaver [9] proposes architectural support to facilitate sleep state by delaying and preempting requests that create common idle and busy periods across cores of a server. The proposed mechanism requires additional hardware to coordinate the sleep periods across the cores. Differently, TS-Bat only uses off-the-shelf hardware and its effectiveness is evaluated on physical systems with real power measurement. Finally, we note that server energy optimization as proposed in TS-Bat can be integrated with more energy-efficient data center network topologies [14] to boost system energy savings.

VII. Conclusion

In this paper, we proposed *TS-Bat*, a new data center energy optimization framework that judiciously integrates spatial and temporal job batching to save energy for multi-core data center servers while meeting QoS constraints. TS-Bat performs global job batching and effective scheduling onto a minimal subset of servers such that opportunities for servers to take advantage of processor package-level low-power modes can be maximized. We implemented a prototype of TS-Bat on a physical testbed with a cluster of servers and evaluated TS-Bat using a variety of workloads. Our results show that temporal batching alone achieves 49% CPU energy savings compared to the baseline configuration without batching. Through combining temporal and spatial batching, TS-Bat achieves up to 68% processor energy savings under various QoS constraints.

VIII. Acknowledgments

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

References

[1] J. Koomey, "Growth in data center electricity use 2005 to 2010," *A report by Analytical Press*, 2011.

[2] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM International Symposium on Computer Architecture*, 2007.

[3] 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.

[4] C. Isci, S. McIntosh, J. Kephart, R. Das, J. Hanson, S. Piper, R. Wolford, T. Brey, R. Kantner, A. Ng, J. Norris, A. Traore, and M. Frissora, "Agile, efficient virtualization power management with low-latency server power states," in *ACM International Symposium on Computer Architecture*, 2013.

[5] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A dual delay timer strategy for optimizing server farm energy," in *IEEE International Conference on Cloud Computing Technology and Science*, 2015.

[6] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *ACM International Symposium on Computer Architecture*, 2014.

[7] 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.

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

[9] D. Meisner and T. F. Wenisch, "DreamWeaver: architectural support for deep sleep," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[10] HP, Intel, Microsoft, Phoenix, Toshiba, "Advanced configuration and power interface specification." http://www.acpi.info/.

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

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

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *ACM International Conference on Parallel Architecture and Compilation Techniques*, 2008.

[14] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A comparative analysis of data center network architectures," in *IEEE International Conference on Communications*, 2014.