WASP: Workload Adaptive Energy-Latency Optimization in Server Farms using Server Low-Power States

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

Abstract—**With the growing energy demands from server farms, it becomes necessary to understand the tradeoffs between energy consumption and application performance. Typically, server farms are provisioned for peak load even when they are mostly operating at low utilization levels. This results in wasteful energy consumption. At the same time, application workloads have Quality of Service (QoS) constraints that need to be satisfied. Optimizing server farm energy consumption with QoS constraints is a challenging task since the workload can have variabilities in job sizes, job arrival patterns and system utilization levels.**

In this paper, we present *WASP*, where we explore techniques that make smart use of the processor and system low-power states, and orchestrate their use with workload adaptivity for more effective energy management. We perform an extensive study of Energy-Latency tradeoffs with simulations, and evaluate WASP on a testbed with a cluster of servers. Our experiments on real systems show that WASP achieves up to 57% energy reduction over a naive policy that uses a shallow processor sleep state when there are no jobs to execute, and 39% over a delay-timer based approach while maintaining the 90th percentile job service latency to be under 2× job execution time.

Keywords-**Cloud Computing; Data Center Energy Optimization; Energy-Latency tradeoffs; Processor and System low-power states; Workload adaptivity**

I. Introduction

Large-scale server farms and data centers account for nearly 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 [2], [3]. The disproportionality in server utilization versus energy consumption occurs largely because of ineffective system-wide energy management and server over-provisioning without understanding or even considering the workload characteristics. Therefore, exploring a *workload-aware* framework that performs effective system-wide energy management, is essential in server farms.

While server-level energy optimization strategies have been studied previously [4], [5], [6], system-level energy management that can automatically adapt to different applications with QoS constraints are ever-more important. Techniques such as Dynamic Voltage/Frequency Scaling (DVFS) only save active power, and do not take advantage of situations where servers can enter low-power states to achieve both static and dynamic energy savings [7]. In other words, DVFS can be useful only in applications that require the servers to be active at all (or most) times. For applications that spawn independent tasks (e.g., web servers), it may not be necessary to keep all servers active especially when the service requests are relatively infrequent. This necessitates a more aggressive energy-saving approach.

In this paper, we present WASP, an energy optimization framework for server farms that is able to adapt itself to meet the performance demands (QoS constraints) while minimizing the system energy through *dynamically* adjusting its parameters based on workload characteristics such as job size, arrival pattern and system utilization. To achieve energy optimization, we explore the use of processor and system low-power states combined with adaptive techniques to orchestrate the entry and exit from these low-power states. WASP also considers variations in job arrival rates for bursty workloads where local spikes in the arrival patterns need to be monitored. This information can guide WASP to provision servers in shallow sleep states such that they can be woken up faster and meet the QoS constraints for tasks.

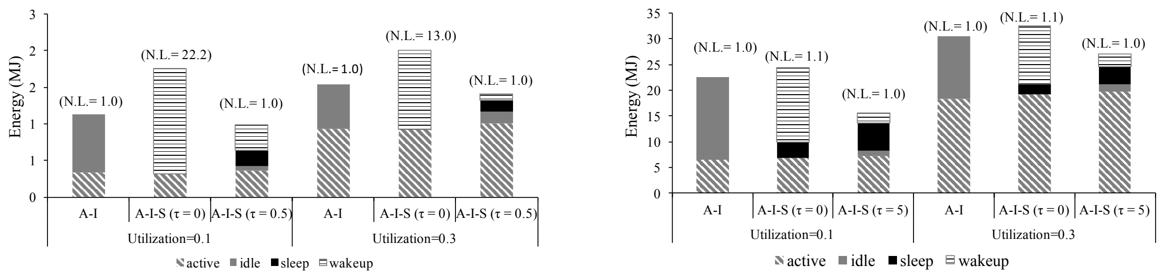
In summary, the contributions of our work are:

1) We motivate the need to jointly optimize for energy-latency in server farm applications in an adaptive manner. We posit that future server farms will need to cater to a variety of workload patterns, and perform energy optimization subject to users' performance demands.

2) We design *WASP*, a novel energy management framework that orchestrates judicious use of the system energy-saving features based on system-wide workload characteristics. We explore the energy-latency Pareto-optimal space in server farms under different system utilization levels and workloads, which is then used by WASP for parameter selection.

3) We investigate the applicability of our adaptive energy-saving schemes to different job arrival patterns: 1. random arrivals modeled by Poisson Process, and 2. non-bursty and bursty real system traces. A subset of servers in shallow sleep states is dynamically provisioned to accommodate spikes in job arrivals.

4) We perform experimental evaluation on a web server testbed, and measure system energy. With a QoS constraint of 90^{th} percentile normalized latency to be under 2× the job execution time on real systems, WASP exhibits up to 57% energy savings over a naive policy that uses a shallow processor low-power state during inactive periods, and up to 39% energy savings over a delay-timer based approach where the processor enters low-power state only after observing system inactivity for a certain delay period.



(a) Workload#1 (small-latency job) (b) Workload#2 (large-latency job)

Fig. 1: Energy breakdown for various workloads in A-I and A-I(τ)-S configurations using simulation. The latency, normalized with respect to A-I configuration (N.L.), is shown on top of each bar.

II. Background

A. Low-power States and Power Model

Low-power states are now an important feature that is widely supported in today's computing systems. The Advanced Configuration and Power Interface (ACPI) [7] specifies various processor low-power states *Cx* (e.g., C1, C6), and system low-power states *Sx*. A higher level C state or S state typically indicates more aggressive energy savings but longer wake-up latencies. For a multi-core processor, low-power states are supported at both core and package levels. When there is no job to execute on a core, it becomes idle and could reside in a *Core C state*. When all cores become idle, the entire processor/package would be resolved to a certain *package sleep state*, which further reduces power. Although processor low-power states can significantly reduce the power consumed by the processor, the server still consumes a considerable amount of power as the platform may still remain active. As a result, in order to achieve even greater energy savings, *system sleep states*, that put platform components into low-power states, are used.

B. Server and Job Model

We model the server farm as a multi-server system that can process multiple jobs (up to the total number of cores) at a time. *Utilization level factor* is defined as the product of the job arrival rate and the average job execution time, which is also the fraction of the time that the server is expected to be busy. We assume that a system-wide load balancer dispatches jobs to the servers within the server farm. The *job latency* is defined as the time elapsed between when a job arrives and when it finishes execution and departs the server. In this paper, the 90^{th} percentile job latency is considered as the QoS target.

C. Smart Use of Low-Power States

Processor/system low-power states help reduce energy consumption by shutting down various processor components and putting devices to power saving mode, especially when the processor utilization levels are low. However, they can also introduce significant performance degradation due to extended wakeup latencies, especially from deep sleep states. Typically, delay timers are utilized to prevent the servers from going to deep sleep prematurely and avoid expensive wakeup latencies. To illustrate the benefits of using sleep states and delay timers for energy savings, we perform simulation experiments that study energy consumption of server farms with 100 servers under two workload settings, a small-latency job (about 5ms) and a large-latency job (about 200ms). We evaluate using two power management policies described below.

1) *Active-Idle*, denoted as A-I, is a power configuration where the server alternates between active and shallow package sleep state, C1. A server is active when at least one of the cores in the processor within the server has a job to process. The server enters C1 if none of its cores are actively running jobs.

2) *Delay-Doze*, denoted as A-I(τ)-S, is a power configuration where the server transitions between three states – active, package sleep C6 (I), and system sleep (S). When all cores are idle, the server immediately enters *I*, then goes to *S* after a delay of τ seconds. If a new job arrives before the delay timer expires in *I*, the server would transition back to the active state.

Figure 1 shows the energy breakdown of a server farm using the above two policies. For each workload, we study both configurations for system utilization levels of 0.1 and 0.3. We make the following observations: **1. Utilizing package and system sleep states can bring energy savings for both workloads**. For example, at utilization level of 0.1, we observe as much as 13% energy reduction in Workload#1 and 29% energy reduction in Workload#2 with A-I(τ)-S compared to the corresponding A-I configurations. **2. Processor and system low-power states have to be used judiciously**. If sleep states are used too aggressively, they may considerably deteriorate the tail latency and waste more energy. Figure 1 shows the energy for A-I(0)-S (where we enter system sleep mode when idle) is higher and the normalized latency is worse compared to Active-Idle for both workloads. The results show that leveraging low-power states in a smart way has good potential in saving server farm energy.

III. Workload Adaptation

While delay timers are useful in saving energy, we note that they lack workload awareness. Large values of delay timers could cause the system to remain in higher power states for longer than necessary resulting in increased energy consumption, while too small values for delay timers result in

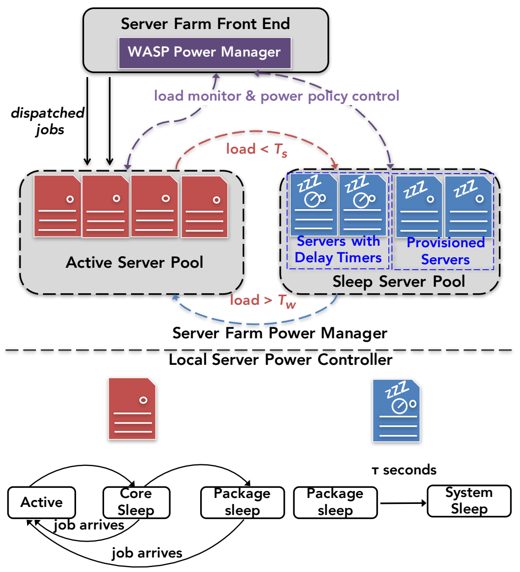


Fig. 2: WASP Power Management Framework

premature entry into low-power states. This can be problematic on two counts: 1) The transition energy between power states can be high. 2) The wakeup latencies from low-power states can degrade system performance.

To incorporate workload-awareness, we explore a two-level adaptive strategy that controls the active and low-power state transitions using a local server power controller and a global server farm power manager.

A. *WASP: Workload-Adaptive Algorithm*

We now present the design for our WASP framework. As shown in Figure 2, the server farm power manager in the front end monitors the current load (number of pending jobs per server) and sends control commands to the local power controller. The server farm power manager puts the servers in either active or sleep modes. The bottom part of Figure 2 presents state transitions coordinated by the local server power controllers.

TABLE I: Notations in WASP power management algorithm

Symbol	Description
V_{act}	servers in active/package sleep mode
V_s	servers in system sleep mode
T_s	workload threshold to reduce active servers
T_w	workload threshold to increase active servers
N_p	number of provisioned shallow sleep servers
τ	delay time before entering system sleep

WASP automatically activates servers when the pending load becomes too high (that could lead to higher average job latency), and then places servers in low-power sleep mode to conserve energy when the workload becomes light. We achieve our goal of balancing energy consumption and latency by estimating the current load and placing servers in different power modes. There are two important parameters in Algorithm 1 that govern transitioning between active and

Algorithm 1: Global Server Farm Power Manager

```
Input: $T_s$, $T_w$, $N_p$, $\tau$, $n$ (total number of servers)
Initialization: $V_{act} = \{s_1, s_2, ..., s_n\}$; $V_s = \{\}$;
while there are unfinished jobs do
    if a new job $j$ arrives at time $t_a$ then
        compute load_per_active_server;
        if $load\_per\_active\_server > T_w$ and $|V_s| > 0$ then
            retrieve a server $s$ from $V_s$;
            $V_{act}$.add($s$);
            create a trans_to_active_mode request tta_r;
            send tta_r to server $s$'s power controller;
    if a job $j$ finishes at time $t_d$ then
        compute load_per_active_server;
        if $load\_per\_active\_server < T_s$ and $|V_{act}| > 0$ then
            retrieve a server $s$ in $V_{act}$;
            $V_s$.add($s$) ;
            create a trans_to_sleep_mode request tts_r;
            if count of shallow sleep servers> $N_p$ then
                tts_r.enableDelayTimer($\tau$);
            else
                tts_r.enableDelayTimer(infinity);
            send tts_r to server $s$'s power controller;
```

sleep modes: 1. T_s, workload threshold per active server below which WASP will put an active server to sleep, and 2. T_w, workload threshold per active server above which WASP will wake up an inactive server.

Global Server Farm Power Manager: All servers are initially in the shallow low-power state, and arriving jobs are placed in the server's local job queue. As jobs arrive, the *load per active mode server* is computed dynamically by the power manager in the front end based on the number of jobs sent to individual servers and the completed jobs. The global server farm power manager maintains lists of servers in active and sleep modes. When new jobs arrive, it first checks if current load per active server is above T_w. If so, it selects a server in sleep server pool (if available) and sends a power mode transition request to the active state for that server. When the load per active server falls below T_s, the power manager selects an active server and sends a power mode transition request to enter sleep mode. Algorithm 1 describes the WASP power manager with its notations shown in Table I.

Local Server Power Controller: The processor transitions to package sleep state when it becomes idle, and stays in that state until it receives the request to wakeup from the global power manager. If the server receives a request for transition to sleep mode, it will first finish up all the pending jobs in the local queue and then enters package sleep after which a delay timer is started. The server enters system sleep upon delay timer expiration. However, if the scheduler chooses to wake up the server before timer expiration (e.g., due to sudden load increase), the timer is reset and the server goes back to active mode.

For large server farms, we can adopt one of two possible solutions: 1) Adopt a distributed power management approach where energy is optimized within individual domains of servers with their own power managers. 2) Adopt a hierarchical solution with multiple levels of global power managers. We note that a distributed power management approach may be more scalable with lower implementation complexity compared to a hierarchical approach that may involve longer latencies for decision making and higher bookkeeping overheads for the servers.

B. Adaptive Server Provisioning

Job arrival pattern may have local spikes (bursty), during which the service latency may suffer, especially when the servers are in low power modes. To mitigate this problem, we provision a subset of servers in shallow sleep states dynamically by setting their delay timer values to infinity. WASP determines the number of provisioned servers *dynamically* through measuring the current standard deviation in the job arrival rate observed over a period of 2 minutes. Specifically, the server provision module samples the number of arrivals and calculates the utilization for each sample period (one second in our current setting). It then uses the sampling window to determine the standard deviation in the level of system utilization. The module will provision $\alpha \times stdev \times number\_servers$ dynamically in shallow sleep state. α is a tunable parameter. By default we set it to 3.0, since it typically covers a vast majority of the population (e.g., more than 99% of the population in Gaussian distribution).

IV. Experimental Setup

We perform two sets of experiments: 1. simulations to explore the Pareto-optimal energy-latency tradeoff as well as corresponding T_s, T_w and τ settings, and 2. prototype implementation on a testbed with web server deployment. In this section, we elaborate on the experimental setup for both approaches.

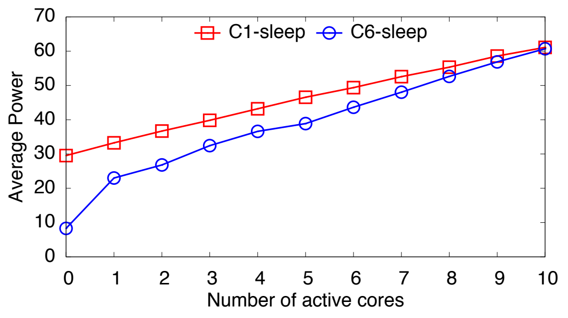


Fig. 3: Power profile of a 10-core Xeon E5 processor with C0-C1 and C0-C6 transition settings whenever the server is idle.[1]

A. Processor Power Profile and Server Power Model

We profile the power consumption of the Intel Xeon E-5 processor [8] using Intel's *Running Average Power Limit* (RAPL) interface. We build a customized cpuidle governor that allows specified low-power state transitions. The processor is programmed to transition between active state (C0) and low-power state *Cx* (e.g., C1, C6). Figure 3 shows the measured power consumption of the processor for two configurations: *C0-C1*, *C0-C6* and at utilization levels from 0% to 100%. Using linear regression, a power model is built for the processor based on the sleep state selection and the number of active cores at full utilization. Table II shows the power consumption when a certain state is chosen for sleep mode. Table III shows the wakeup latencies for various low-power states. Note that the processor sleep state transition latencies are reported by the Linux cpuidle driver [10].

[1]We use a microbenchmark that can calibrate itself and occupy the core based on required utilization settings. To occupy multiple cores, we run multiple copies of this microbenchmark each pinned to a core.

TABLE II: Power (W) breakdown for a system with n_a active cores

Component	Core sleep C1*	Core sleep C6 †	Pkg. sleep C6	System sleep
CPU	33.0+3.1 $\times (n_a - 1)$	23.0+3.8 $\times (n_a - 1)$	8.3	8.3
RAM [8]	10.8	10.8	4.9	1.4
Platform [9]	45.5	45.5	23.6	4.8
Total Power	89.3+3.1 $\times (n_a - 1)$	79.3+3.8 $\times (n_a - 1)$	36.8	14.5

$*$ processor is active and the rest of the idle cores are in *C1* state.
† processor is active and the rest of the idle cores are in *C6* state.

TABLE III: Processor/System low-power states and wakeup latencies

Low-power State	Wake-up latency
core sleep C1	10 μs
core sleep C6	82 μs
package sleep C6	1 ms
system sleep	5 s

B. Simulation Platform

WASP uses an event-driven simulator based on Bighouse [11] that models server farm workloads and multi-server activity. We simulate a server farm with 100 ten-core servers (by default). In all of our experimental results, we report the steady state statistics by disregarding the warm-up period of the first 10,000 jobs. In the simulation, we use short latency (Web service-like) jobs with $s = 4.2ms$ and long latency (DNS service-like) jobs with $s = 194ms$ as representatives based on prior studies [9]. For each of the representative workloads, we generate synthetic job arrivals with different utilization levels (0.1 for low, 0.3 for average [2], and 0.6 for high). Random job arrivals are modeled by Poisson process [5]. Besides synthetic workload, we also perform simulation based on Wikipedia traces.

C. Real System Experiments on Testbed

We deploy a testbed with a cluster of 10 application servers together with one load-generating server and one load-balancing server; all servers support Intelligent Power Management Interface (IPMI) interface [12] for system-level power monitoring. Each application server is configured with the apache web service. The load generator keeps sending web requests to the system according to real system traces (See Section VI for further details).

V. Energy-Latency Tradeoff

We conduct parameter exploration with thousands of simulator runs for every workload at a given utilization level to find optimal ranges of T_s, T_w and τ values under various QoS constraints. We study using two synthetic workloads at three different system utilization levels, and using real system traces from Wikipedia. In each run, we measure energy consumption, average job latency, and 90^{th} percentile normalized job latency. The energy-latency frontier curves are then generated by eliminating the less-desired pairs which do not lead to Pareto-optimality. We note that further opportunity may exist if we combine sleep states with DVFS, that reduces dynamic power. Therefore, we conduct exploratory experiments to study the benefits in combining WASP with DVFS to optimize the energy spent during job execution.

A. Frontier Curves on Random Arrivals

Figure 4 presents the relationship between energy and job latency under various frequency (f) settings.

WASP shows good energy-latency tradeoffs across workloads and utilization levels. In Web Service workload at utilization level of 10%, without any frequency scaling (f=1.0 is depicted as solid red curves), WASP demonstrates 57% reduction in energy compared to a naive Active-Idle (A-I) power management policy (See Section II-C). Note that the 90^{th} percentile normalized latency is within 2.0. At utilization level of 30%, WASP achieves 39% energy reduction for Web Service workload. We note that WASP shows a similar trend in other workloads as well.

Using DVFS in conjunction with WASP improves Pareto optimality to a limited extent with increased tail latency. Our results show that, by lowering f to some extent, higher energy reduction can be achieved without adversely affecting the latency values. For example, for DNS Service at the high utilization of 0.6 with $f = 1.0$, the energy reduction can be up to 17% when the 90^{th} percentile normalized latency is 2.0. When $f = 0.7$, energy decreases by another 6% with tail latency degrading to 3.0. However, if f is lowered too much, there is a significant deterioration of performance without a proportional increase in energy savings. Consequently, to achieve higher energy savings in the active mode using DVFS, appropriate selection of f is needed. While better energy saving may be possible by incorporating more intelligent DVFS control algorithms, we note that the room for improvement is limited due to the fact that WASP already takes advantage of system idleness for energy improvement.

B. Frontier Curves on Non-bursty Traces

We obtain publicly available Wikipedia website traces [13] that include arrival timestamps for each web request along with the URL. Studies by van Baaren et al. [13] have characterized the average job service time as 3.5 ms. For our study, we obtain the arrival timestamps from the traces, and adopt the job service time distribution from [13]. We simulate a one-hour Wikipedia trace on a 10-machine configuration. Figure 5 shows the Pareto-optimal curves for energy and latency. WASP is able to achieve 58% energy saving over Active-Idle with 90^{th} percentile normalized latency below 2.0, which is similar to the energy reduction observed in the synthetic Web Service workload in Section V-A.

C. WASP Parameters

A natural question that arises during energy optimization is: what set of parameter values in the WASP algorithm help achieve energy-latency Pareto-optimality? Understanding the characteristics of these parameter values is essential for users to dynamically configure the system under various workloads and latency constraints. For each workload, we collect Pareto-optimal values of T_s, T_w, and τ for different utilization levels. Due to space limitations, we are unable to show all of our results. We list our key observations below:

1) *The values of τ that lead to different normalized tail latencies are fairly independent of utilization levels, but is job-size dependent*. The Pareto-optimal τ for Web Service is 0.5s and τ increases with job size, e.g., 10s for DNS Service.

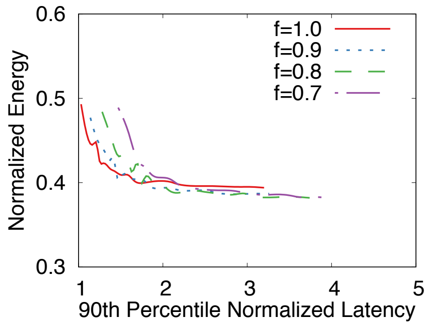
2) *The values of T_w are also independent of utilization levels*. Intuitively, T_w controls how fast a server in sleep mode would transition to active mode. As T_w increases, servers are woken up less often, which saves energy at the cost of increased tail latency. This gives hints on why T_w also scales linearly with latency values.

3) *T_s values are independent of job execution latencies and utilization levels*. Moreover, when upper bound of the 90^{th} percentile normalized latency is set to stringent values such as 2.0, the values of T_w and T_s are also quite close across benchmarks.

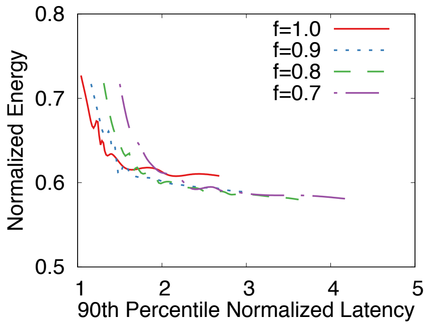
The characterization of the WASP parameters helps to optimally select T_s, T_w and τ parameters according to the workload, utilization levels and tail latency requirements in an *automated* manner. Our experiments show that our regression model can accurately predict the three parameters fairly quickly. For verification, we use the regression-derived parameters (for a specific QoS) and compare the tail latency and energy consumption with the ones we got from the frontier curves, and our results showed less than 5% absolute error.

VI. Evaluation on Real System

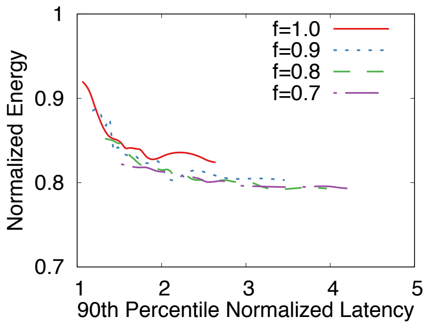
We evaluate WASP on a testbed with 10 Dell Poweredge servers equipped with Inten Xeon-based processors with all of the servers deployed on a dedicated rack. We installed a modified version of the Apache HTTP server for our Local Power Controller. We extended the Local Power Controller to also include a Delay-Doze timer. The Global Server Farm Power Manager is added to an additional apache server with the *mod_proxy_balancer* module used for load balancing. Specifically, the load balancer performs operating mode transitions in servers (as discussed in Section III-A); this is done by sending special HTTP requests (*/hostname/trans-to-active-mode/*, */hostname/trans-to-lp-mode*) to the application server. It also monitors the power state for each server, and manages the server wakeups (from system sleep) using IPMI interface supported by Dell systems. The special requests are handled



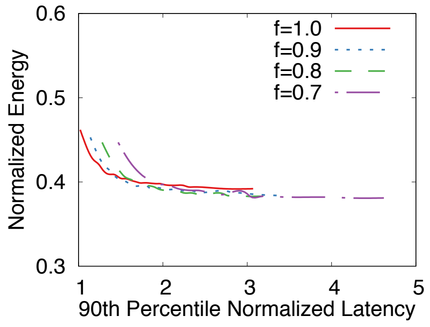
(a) Web Service, Util=0.1



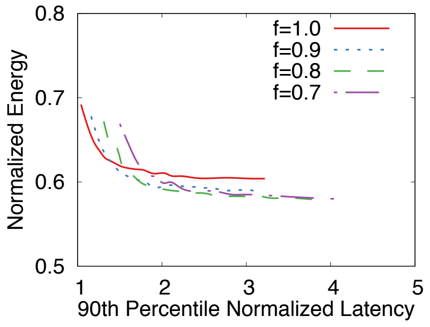
(b) Web Service, Util=0.3



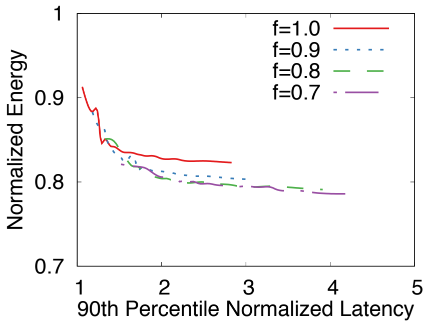
(c) Web Service, Util=0.6



(d) DNS Service, Util=0.1



(e) DNS Service, Util=0.3



(f) DNS Service, Util=0.6

Fig. 4: Pareto-efficiency frontier curves for energy vs. latency under different frequency settings (f). Job arrivals are modeled as a Poisson Process, and energy, latency and frequency values are normalized with respect to the configuration using Active-Idle power management policy.

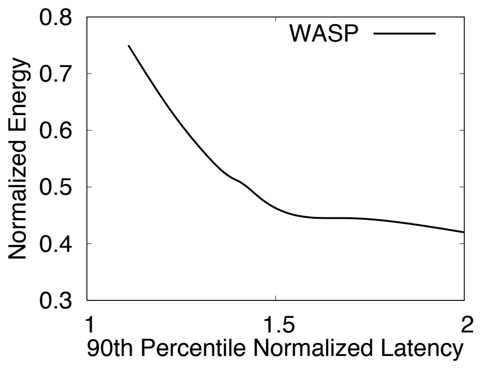


Fig. 5: Pareto-efficiency frontier curve for Energy-Latency tradeoffs on real-world Wikipedia traces.

by the local power controller that would determine the server low-power transitions accordingly. We set up a custom cpuidle governor which allows direct processor C-state transitions from userspace (e.g., C0-C6). For power measurement, we leverage two techniques: the RAPL interface for fine-grained component power, and the IPMI's system power management interface for coarse-grained server power. We evaluate the effectiveness of WASP by providing two sets of workloads to the system: the non-bursty Wikipedia workload, that does not require server provisioning, and four bursty NLANR workloads [14], that require server provisioning to handle bursty workloads (See section III-B).

A. *Wikipedia Trace*

We performed real system energy measurements by deploying Wikipedia software stack, namely Wikipedia application (Mediawiki), database system (Mysql) on servers. We compare WASP against Active-Idle and Delay-Doze approaches described in Section II-C. To capture detailed energy breakdown, we leverage RAPL interface for fine-grained power measurement. The RAPL utility records the CPU and RAM power values periodically. We configure WASP with the T_s, T_w and τ parameters that achieve energy-latency Pareto-optimality with tail latency constraint set to 2.0. Similarly, for Delay-Doze, we explore various values of the delay-timer and choose the setting that achieves the best power with the same tail latency constraint. From our experiments, we get actual CPU and RAM energy consumption for each server. To get the overall server energy, we also factored in the platform energy shown in Table II.

Figure 6 shows the per-server energy breakdown in terms of CPU, DRAM, and platform energy. With Active-Idle power management, all the 10 servers have similar energy consumption. With Delay-Doze, some of the servers are able to stay in system sleep state for longer periods of time, thus saving energy. With WASP, we can clearly see that most of the servers drastically reduce energy consumption, and only a minimal subset of servers (server#6 and #10) are used for servicing jobs. Note that the energy consumption of server#10 is slightly higher than that of Active-Idle power management

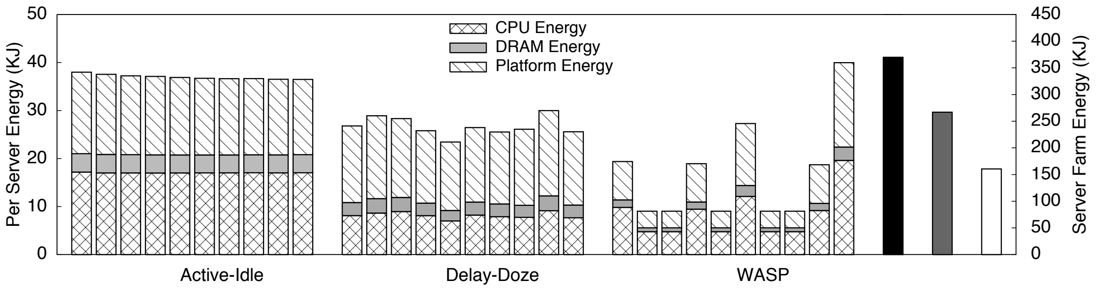


Fig. 6: Energy measured on a server farm with 10 servers with different energy management policies. The first three groups of bars represent energy breakdown in each server when Active-Idle, Delay-Doze and WASP are applied, respectively. The rightmost three bars illustrate the total server farm energy consumption for Active-Idle (black bar), Delay-Doze (gray bar) and WASP respectively (white bar).

since the server is at a higher utilization level while other servers remained inactive. Overall, WASP gains 39% reduction in energy saving compared to Delay-Doze, and 56% energy savings compared to Active-Idle.

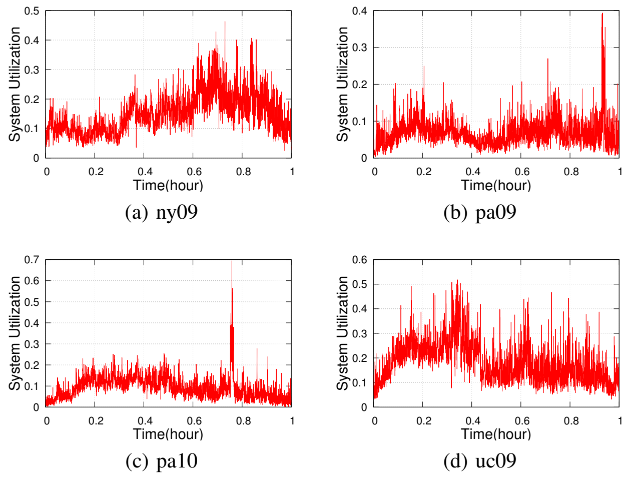


Fig. 7: System utilization for four bursty traces.

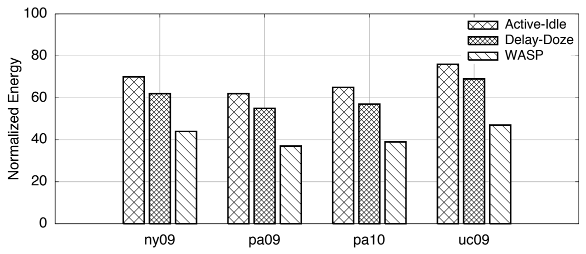


Fig. 8: Normalized energy consumption relative to peak energy on a 10-server cluster

B. Bursty Traces

As the raw NLANR network traces [14] present job arrivals that are too infrequent for the server farm system with 10 application servers (less than 2%), we speed up the trace by scaling the 24-hour trace to one hour. We choose four traces, namely *ny09*, *pa09*, *pa10* and *uc09*. Figure 7 shows the utilization levels (scaled) for the four traces over one hour. All traces exhibit bursty traffic patterns. For example, the *ny09* trace has highly fluctuating utilizations ranging from 4% to 45% with a large number of spikes. To run the traces, we set up a software stack similar to the one in Section VI-A. Each request in the trace is serviced by a PHP script that accesses a pre-defined set of pages randomly, and we note that the average service time is about the same as Wikipedia web requests.

To enable server provisioning, the Server Farm Power Manager additionally samples the server farm utilization levels based on the job arrival rates. Utilization is calculated as the product of job arrival rate and average job execution time. Standard deviation on the samples for utilization levels is calculated every 120 seconds. The number of provisioned servers is calculated dynamically (See Section III-B for details). Note that, in our comparative studies, the delay-timer values are re-evaluated for each trace such that best possible energy savings are had while meeting the QoS constraints. Figure 8 shows the energy consumption for the four bursty workloads using Active-Idle, Delay-Doze and WASP. The energy is normalized to the peak energy which is $PeakPower * Time$. The energy reduction for WASP ranges from 34% to 40% compared to Active-Idle. Even with the best delay timer settings, Delay-Doze only achieves 9% to 12% energy reduction in bursty workloads. We observe that due to the job arrival rate spikes (especially for *uc09*), in order for Delay-Doze to meet the tail latency constraint of 2.0, the delay timer has to be set to larger values, and in turn the servers have limited chance to enter system sleep state.

VII. Related Work

Techniques to improve application and network energy efficiency have been well studied in the literature [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. While DVFS-based mechanisms have been proposed to optimize processor power [16], [17], they are less effective for systems in low utilization when idle power is dominant. As a result, prior works [5], [9] propose architectural support to facilitate sleep state management on multi-core servers that include scheduling policies to delay, pre-empt and execute requests, and artificially create idle and busy periods across cores of a server.

Lo et al. [25] leverage *RAPL* to dynamically adapt the runtime power of data center according to job latency feedback. The trends in server energy proportionality have been studied by Ryckbosch et al. [26]. Sleepscale [4] utilizes speed scaling and server sleep states jointly to reduce the average power for a single server. We note that other approaches, such as Knightshift [6], explore more specialized approaches such as exploiting heterogeneity of processor cores to improve energy. The model has been extended by Wong et al. [27] 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. Through combining our proposed approach with the energy improvement solutions on heterogeneous servers, we can further boost energy savings.

Gandhi et al. [21], [28] have proposed a delayed-off mechanism that turns off a server after it is idle for a preset period of time. Autoscale [28] reduces multi-server system power by controlling the number of active servers while satisfying the QoS. In our study, we show how system sleep states can be smartly utilized instead of physically turning a server off since this may introduce unacceptable spikes in job latencies, especially for small jobs. Studies by Kanev et al. [23] highlights the need for comprehensive sleep state selection. Our work shows that in order to improve energy-latency tradeoff, besides the selection of sleep state, a smarter management of sleep states and their transitions is equally important.

VIII. Conclusion

In this paper, we explored techniques that makes smart use of processor/system low-power states and orchestrate them adaptively with changing workloads for effective energy management. We propose a two-level power management framework that features a global server farm power manager and a local server power controller. We performed an extensive exploration of Pareto-optimal Energy-Latency tradeoffs. The results show considerable energy savings on different synthetic and real workloads. Our experimental results on real systems show that WASP achieves up to 57% energy saving over a naive policy that only uses a shallow processor sleep state, and 39% saving over a delay timer based approach, while keeping the 90^{th} percentile latency to be under $2\times$ average job service time.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under CAREER Award CCF-1149557 and CNS-1320226.

References

[1] R. Brown, "Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431," *Lawrence Berksleley National Laboratory*, 2008.

[2] X. Fan, W.-D. Weber, and L. A. Barroso, "Power Provisioning for a Warehouse-sized Computer," in *ACM SIGARCH Computer Architecture News*, ACM, 2007.

[3] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware Cluster Management," in *Proceedings of Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, 2014.

[4] Y. Liu, S. C. Draper, and N. S. Kim, "SleepScale: Runtime Joint Speed Scaling and Sleep States Management for Power Efficient Data Centers," in *Proceedings of Intl. Symp. on Computer Architecture*, IEEE, 2014.

[5] D. Meisner and T. F. Wenisch, "DreamWeaver: Architectural Support for Deep Sleep," *ACM SIGPLAN Notices*, 2012.

[6] D. Wong and M. Annavaram, "KnightShift: Scaling the Energy Proportionality Wall through Server-level Heterogeneity," in *Proceedings of International Symposium on Microarchitecture*, 2012.

[7] HP, Intel, Microsoft, Phoenix, Toshiba, "Advanced Configuration and Power Interface Specification," 2015.

[8] Intel, "Intel Xeon Processor E5-1600/E5-2600/E5-4600 Product Families," 2012.

[9] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: Eliminating Server Idle Power," in *Proceedings of Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

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

[11] D. Meisner, J. Wu, and T. F. Wenisch, "BigHouse: A Simulation Infrastructure for Data Center Systems," in *Proceedings of International Symposium on Performance Analysis of Systems & Software*, 2012.

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

[13] E.-J. van Baaren, "Wikibench: A Distributed, Wikipedia based Web Application Benchmark," *Master's thesis, VU University Amsterdam*, 2009.

[14] "The Univ. of Waikato NLANR Projects," 2012. http://www.nlanr.net.

[15] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A Comparative Analysis of Data Center Network Architectures," in *Proceedings of International Conference on Communications*, IEEE, 2014.

[16] S. Herbert and D. Marculescu, "Analysis of Dynamic Voltage/Frequency Scaling in Chip-multiprocessors," in *Proceedings of Intl. Symp. on Low Power Electronics and Design*, ACM, 2007.

[17] D. C. Snowdon, S. Ruocco, and G. Heiser, "Power Management and Dynamic Voltage Scaling: Myths and facts," in *Proceedings of Workshop on Power Aware Real-time Computing*, 2005.

[18] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A Dual Delay Timer Strategy for Optimizing Server Farm Energy," in *Proceedings of Intl. Conf. on Cloud Computing Technology and Science*, IEEE, 2015.

[19] J. Chen and G. Venkataramani, "A hardware-software cooperative approach for application energy profiling," *IEEE Computer Architecture Letters*, vol. 14, pp. 5–8, Jan 2015.

[20] J. Chen and G. Venkataramani, "enDebug: A Hardware–software Framework for Automated Energy Debugging," *Journal of Parallel and Distributed Computing*, vol. 96, pp. 121–133, 2016.

[21] A. Gandhi and M. Harchol-Balter, "How Data Center Size Impacts the Effectiveness of Dynamic Power Management," in *Proceedings of Allerton Conf. on Communication, Control, and Computing*, IEEE, 2011.

[22] J. Chen, F. Yao, and G. Venkataramani, "Watts-inside: A Hardware-software Cooperative Approach for Multicore Power Debugging," in *Proceedings of Intl. Conf. on Computer Design*, IEEE, 2013.

[23] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, "Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications," in *Proceedings of Intl. Symp. on Workload Characterization*, IEEE, 2014.

[24] 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.

[25] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards Energy Proportionality for Large-scale Latency-critical Workloads," in *Proceedings of Intl. Symp. on Computer Architecture*, IEEE, 2014.

[26] F. Ryckbosch, S. Polfliet, and L. Eeckhout, "Trends in Server Energy Proportionality," *Computer*, vol. 44, no. 9, pp. 69–72, 2011.

[27] D. Wong and M. Annavaram, "Implications of High Energy Proportional Servers on Cluster-wide Energy Proportionality," in *Proceedings of Intl. Symposium on High Performance Computer Architecture*, IEEE, 2014.

[28] 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*, vol. 30, no. 4, p. 14, 2012.