2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)

μDPM: Dynamic Power Management for the Microsecond Era

Chih-Hsun Chou[†*] Laxmi N. Bhuyan[§] Daniel Wong[†]

[†]*Department of Electrical & Computer Engineering*
University of California, Riverside
Email: cchou011@cs.ucr.edu, dwong@ece.ucr.edu

[§]*Department of Computer Science & Engineering*
University of California, Riverside
Email: bhuyan@cs.ucr.edu

***Abstract*—The complex, distributed nature of data centers have spawned the adoption of distributed, multi-tiered software architectures, consisting of many inter-connected microservices. These microservices exhibit extremely short request service times, often less than 250μs. We show that these "killer microsecond" service times can cause state-of-the-art dynamic power management techniques to break down, due to short idle period length and low power state transition overheads. In this paper, we propose *μDPM*, a dynamic power management scheme for the microsecond era that coordinates request delaying, per-core sleep states, and voltage frequency scaling. The idea is to postpone the wake up of a CPU as long as possible and then adjust the frequency so that the tail latency constraint of requests are satisfied just-in-time. μDPM reduces processor energy consumption by up to 32% and consistently outperforms state-of-the-art techniques by 2x.**

***Keywords*-Dynamic power management, DVFS, Sleep states**

I. INTRODUCTION

With the growth of large-scale distributed systems and platform-as-a-service cloud systems, a new design pattern of software architecture has emerged. These distributed, multi-tiered software consists of numerous interconnected smaller services, popularly called microservices [1]. The simplified functionality of microservices has ushered in the era of microsecond service times. In addition to software applications entering the microsecond era, new breeds of low-latency I/O devices with microsecond access latencies are also emerging [2, 3]. Most recently, Google dubbed this the "era of the killer microsecond" and made a call for computer scientists to design "microsecond-aware" systems stacks as many existing systems are not well-designed for the challenges of microsecond latencies. *This paper explores the implications of application's microsecond service times on state-of-the-art dynamic power management techniques.*

These applications are typically latency sensitive with quality-of-service largely determined by tail latency, not average latency [4]. Servers running latency-critical workloads are usually kept lightly loaded to meet strict tail latency targets, with utilization between 10% and 50% [5–10]. However, this low utilization results in poor server energy efficiency as servers are not energy-proportional and consume significant power at low server utilization [11–16].

[*]Work performed during Ph.D. studies at UCR. Currently at Xilinx, Inc.

To reduce power consumption, modern processors are commonly equipped with two classes of dynamic power management (DPM) mechanisms: *performance scaling* and *sleep states*. Performance scaling, such as dynamic voltage and frequency scaling (DVFS), provides power savings by providing superlinear power savings for linear slowdown in frequency. Unfortunately, the effectiveness of DVFS is diminishing with improved technology scaling as the operating voltage approaches the transistor threshold voltage [17, 18]. While frequency scaling only reduces dynamic power, static power is equally important[19–21]. To reduce static power, sleep states (also called C-states) are designed to save power during idle periods. These sleep states trade sleep/wake-up latency for power savings by powering down different parts of the core (such as core clock, PLL, and caches). As a result, idle power consumption is determined by the C-state that the core enters.

Although servers are usually kept lightly loaded, peak load is considered when setting the proper target tail latency. Due to this, the *observed* tail latency of servers running under low load will be far lower than the *target* tail latency. This "latency slack" that exists between the observed and target tail latency has been exploited by many recent state-of-the-art dynamic power management. For example, DVFS-based techniques [11, 13, 22, 23] and sleep-based techniques [6, 18, 24, 25], have been proposed to slow down processing, or delay processing, so that requests finish just-in-time before the target tail latency. While proven effective, it is unclear how these management techniques will hold up in the microsecond era.

In this paper, we make the following contributions:

- In section II, we present the first exploratory study on the implications of microsecond request service time on existing DPM mechanisms and their limitations. We find that existing DPM schemes break down— specifically, DVFS-based schemes cannot find enough opportunities to slow down, and sleep-based schemes cannot enter a deep enough sleep state to be effective.
- In section III, we propose *μDPM*, a power management scheme for the micro-second era. The key insight driving μDPM is that by carefully coordinating DVFS, sleep and request delaying, we can achieve energy savings where

all others fail. In addition, we additionally introduce a *criticality-aware scheduler* to coordinate scheduling across multi-core in order to keep cores in its current power state to minimize state transition overheads.

- In section IV, we evaluate μDPM under a variety of latency-critical workloads. We show that μDPM reduces CPU energy consumption by up to 32%, and consistently outperforms state-of-the-art techniques, even with micro-second-level request service time constraints.

II. Implications of Microsecond Request Service Time on DPM

Effect on Request Service Times: To identify the effect of microservice applications on request service time, we first explore 4 workloads representing common latency-sensitive microservices. We run these real workloads on a server with Intel Xeon E5 2697-V2 12-core processor to obtain the service time distribution. Workload details are discussed in Section IV. Figure 1 shows the result of our study.

Observation 1: The majority of request service times are less than ~250μs, even with millisecond tail latencies[1]
We define the tail latency similar to [11] as the 95th percentile latency when the application is running at a synthetic medium load without applying any power management. The target tail latency of these workloads is 150μs for Memcached, 800μs for SPECjbb, 1100μs for Masstree, and 2100μs for Xapian. Despite the length of the target tail latency, the actual typical request service time is significantly shorter and often an order-of-magnitude shorter.

Figure 1a shows the observed request service time distribution. In our experiments, we observe that 95% of request service time complete within 33μs for Memcached, 78μs for SPECjbb, 250μs for Masstree, and 1200μs for Xapian. Despite Xapian's long service time tail, a significant 45% of Xapian's request still completes under 250μs. These short service time requests dominate, and lead to high short-term load variability making DPM especially challenging [11].

[1]Carefully note we make a distinction between *service time* and *latency*. Latency includes queueing time and service time.

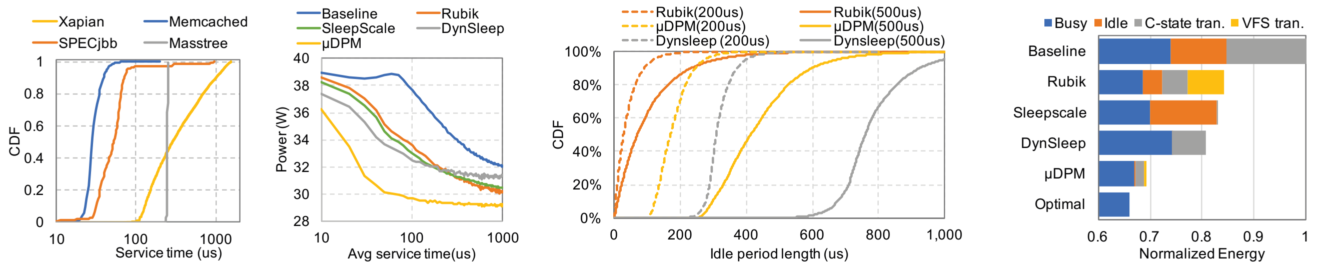
Effect on Dynamic Power Management: We explore various state-of-the-art dynamic power management schemes, such as a DVFS-only technique (Rubik[11]), a coordinated DVFS and Sleep technique (SleepScale[22]), and a Deep Sleep-only technique (DynSleep[24]). All of these prior techniques exploit existing latency slack and are quality-of-service aware to meet tail latency constraints.

Power savings can be achieved by slowing down request processing through DVFS [11, 13, 22, 23, 26] or delaying request processing through Sleep [18, 24]. Due to it's better responsiveness in handling short-term variation, we will evaluate Rubik as representative of other DVFS techniques, such as Pegasus [13], Adrenaline [27], and TimeTrader [23].

While other Deep Sleep-only techniques exist, such as PowerNap [6] and DreamWeaver [18], we observe limited power savings due to the requirement for full-system idleness where the entire processor socket is idle. CARB [25] also leverages sleep states in latency-critical applications by selecting the optimal number of cores to keep on, placing inactive cores into deep sleep. However, an active cores still consume significant idle power due to shallow sleep state selection. Therefore, we evaluate DynSleep [28] as the target Deep Sleep-only mechanism. Unlike PowerNap and DreamWeaver, DynSleep leverage fine-grain per-core sleep states. In addition, these Deep Sleep techniques also utilize request delaying, and is therefore representative of request delaying-only techniques [29].

Due to lack of support for measuring fine-grain power statistics (transition overheads, etc.) in real systems, and lack of hardware control to faithfully implement these DPM techniques in real-hardware (i.e., no per-core DVFS), we perform the remainder of this motivational study through our in-house simulator based on BigHouse stochastic queuing simulation, a validated methodology for simulating the power-performance behavior of data center workloads [30]. Based on our prototype evaluation, we expect the following observed trends are similar in real servers. Section IV provides more details about our evaluation methodology.

C-states overview: Table I shows the empirically measured idle state power and transition overhead from our



(a) Service time dist. (b) Power consumption (c) Idle period length (d) Energy breakdown

Figure 1: (a) Service times are commonly <250μs. (b) In this short service time region, state-of-the-art power management schemes have diminishing effectiveness. Counter-intuitively, deep sleep states provide the most benefits in this region. (c) Short request service time causes fragmented idle periods. (d) Significant energy wasted from idle and transition overheads.

Table I: Measured CPU core C-states.

State	State tran. time	Residency time	Power per core
C0	N/A	N/A	~4W
C1	1μs	1μs	1.43W
C3	59μs	156μs	0.43W
C6	89μs	300μs	~0W

evaluation server, along with the target residency time. CPU cores consume significantly lower power in deeper C-states, along with increasing transition overheads since it takes more time to disable/enable the corresponding on-chip components. During state transitions, the core consumes full power. Thus, the CPU core should only enter a particular C-state only if the idle duration is greater than a threshold, called the target residency time [31], in order to achieve net energy savings. Therefore, C1 is optimal with idle period length of 1-156 μs, C3 with period length 156-300 μs, and C6 with period length 300 μs. The Linux *menu* idle governor estimates the idle period length and selects the best state for that idle period, while the *ladder* governor enters the shallowest state and move to a deeper state if the processor remains in a long enough sleep state. If idle periods are short, as is common with short request service times, then governors will consistently select shallow sleep states.

In the Baseline scheme, the processor operates at maximum frequency and uses the Linux menu governor [32] to select C-states. We similarly use the Linux menu governor to manage sleep states for Rubik (VFS-only), with frequency states updated at every incoming request. SleepScale (VFS+Sleep) selects the optimal C-state and optimal frequency based on historical profiling of idle lengths. Rather than predicting idle period length at every idle event, and frequency at every request arrival, SleepScale sets this at a coarse-grain epoch level (every 60 seconds) to reduce transition overheads, but at the cost of fine-grain opportunities. DynSleep (Deep Sleep) directly enters C6 and runs at maximum frequency when active.

Observation 2: Dynamic power management breaks down at microsecond request service times
Figure 1b shows the average power consumption of various state-of-the-art dynamic power management techniques across different request service times. For this experiment, we simulated an exponential service distribution with varying average service time, shown on the x-axis, at a medium (40%) load. We observed similar trends across other loads. Based on our empirical experiments, we observed a typical target tail latency to service time ratio of 5x, and therefore set the target tail latency at 5x the average service time.

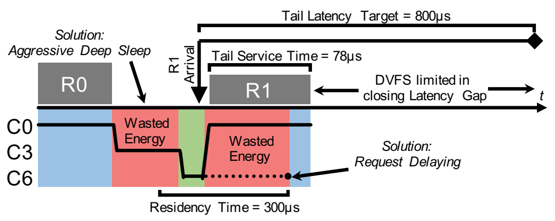
In general, as the request service time decreases, power increases due to fewer opportunities for low power states. In the 250-1000μs range, we observe that dynamic power management schemes that utilize DVFS provide the lowest power due to having ample opportunity to slow down request processing to save power. However, once the average service request times drop below 250μs, both DVFS-based and Sleep-based techniques begin to break down. This is because DVFS techniques cannot handle the short-term variability of short request service times and cannot find enough opportunities to slow down request processing. Meanwhile, Sleep techniques cannot enter a deep enough sleep state due to short idle cycles. *Surprisingly, techniques utilizing Sleep begin to outperform techniques utilizing DVFS!* This can be explained by the trend in idle period lengths.

Observation 3: Short service times fragment idle periods It is well known that *utilization* has a significant impact on sleep opportunities [6, 18]. We additionally find that *request service time* also have a significant impact on sleep opportunities. Specifically, short request service time can fragment idle periods into short idle periods that sleep states cannot take advantage of. Figure 1c shows the idle period length (in μs) under 200μs service time (dotted line) and 500μs service time (solid line). Similarly, we conservatively set the target tail latency to 5x the service time.

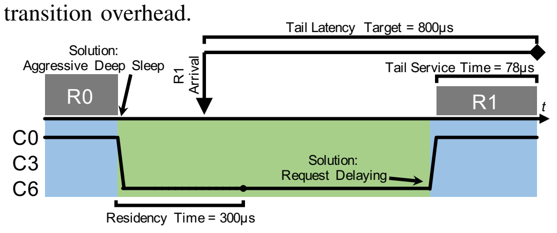
Clearly, as service time decreases, idle period lengths similarly decreases eventually leading to the ineffectiveness of low power states. For example, the baseline curve in Figure 1b can no longer save any power at ~80μs service time due to the inability to enter a deep sleep state. The pervasiveness of shallow sleep states has already been observed in Google's production data center [13, 14].

In Figure 1c, we also observe that DPM schemes can have a great effect on idle period length. Specifically, Sleep-based techniques are able to significantly extend idle period lengths, even under very short service times, by consolidating idle periods through delayed request processing. This can be seen with the lack of short idle periods (<200μs) which is dominant in Rubik. In addition, the idle periods for DynSleep with 200μs service time is at least double that of Rubik at 500μs service time, enabling ample opportunities for sleep states in latency-critical scenarios. *Therefore, the key to sustaining power savings under short service time is by coalescing idle periods into longer idle periods that are better utilized by sleep states.*

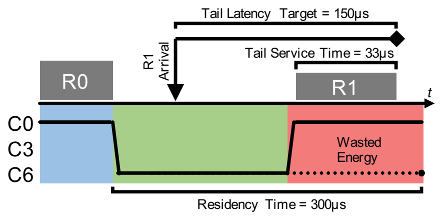
Observation 4: State-of-the-art dynamic power management suffers from significant transition and/or idle power
Existing state-of-the-art power management technique has focused primarily on exploiting latency slack to save power.In our experiments, we found that the energy overheads of idleness and state transition can accumulate and account for a non-trivial fraction of energy consumption. Figure 1d shows the energy consumption breakdown of state-of-the-art power management techniques. For this experiment, we ran a synthetic workload with average service time 80 μs, at the point where Baseline no longer saves power. In this figure, total energy consumption is broken down into 4 parts: (1) energy to process requests (*busy*), (2) energy consumed when a core is idle—can also be in a shallow C-state—and waiting for requests(*idle*), (3) the



(a) Existing DPM limitations include inefficient sleep state management, limited range of frequency states, and significant low-power state transition overhead.



(b) μDPM: Careful coordination of aggressive deep sleep (eliminates idle power), request delaying (meets residency time), and DVFS (just-in-time target tail latency) can achieve energy savings in microsecond workloads.



(c) Under extremely short tail latency targets and service times, μDPM will sacrifice energy savings to meet quality-of-service.

Figure 2: Existing DPM limitations and μDPM overview. Request arrival indicated with ▼. Target tail latency indicated with ◆. (a) and (b) labeled with SPECjbb timing. (c) labeled with Memcached timing. Even with microsecond service times, deep sleep can be utilized when properly coordinated with request delaying and DVFS.

energy spent on C-state transitions (*C-state tran.*) and (4) the energy spent on DVFS transitions (*DVFS tran.*). The energy consumption is normalized to the total energy consumption of Baseline. For comparison, we also include an "Optimal" scenario which runs at the lowest frequency that satisfies tail latency and assumes no idle power or C-state/DVFS transition overheads.

Sleep overheads: Baseline, Rubik and SleepScale all have significant idle energy due to sub-optimal sleep state selection, which reflects our previous observations. This is illustrated in Figure 2(a). When all requests are finished, the Menu governor predicts a C-state to enter (in the figure, C3). Since a shallow sleep state was selected, idle power is still consumed. The processor's power control unit (PCU) can then independently decide to perform a C-state promotion (C3 to C6) if it decides to sleep more aggressively [14]. This behavior can also occur through OS control with the Ladder idle governor. The processor then wakes up when a request arrives. If the processor wakeup occurs before the C-state's target residency time, then overall, that sleep period would have resulted in an increase in energy, rather than a decrease in energy consumption. Dreamweaver partially addressed this issue by batching requests, using a timeout mechanism, to line up system-level idleness and increase sleep period lengths. However, timeouts are not determined per-request and do not fully exploit the latency slack available.

Idle energy accounts for 5% of total energy consumption in Rubik and 15% in SleepScale. SleepScale tends to conservatively predict a shallow C-state every epoch due to coarse-grain updates every 60 seconds. However, this also limits transition overheads to only 5% of total energy consumption. DynSleep is able to directly enter C6 state and delays request processing as long as tail latency constraints are met. Due to this, idle energy consumption is virtually non-existent. However, DynSleep suffers from significant C-state transition overhead as some sleep events may not meet the target residency time (18% of total energy consumption).

Frequency scaling overheads: Due to Rubik's aggressive DVFS reconfiguration at every request arrival, DVFS transition overhead accounts for 10% of total energy consumption. In this paper, we conservatively assume the presence of fast integrated on-chip voltage regulators and use a 10μs DVFS transition time. Many prior works have empirically observed DVFS transition times between 6-70μs [33–37], with our ACPI and cpupower tools reporting 10μs transitions. Even though fast on-chip voltage regulators may switch in 0.5μs, the internal microcontroller of the Power Control Unit [33] spends significant time managing these voltage regulators, resulting in longer transition latency. Because of this non-insignificant transition latency, care must be taken while designing power management scheme with DVFS.

In addition to DVFS transition overheads, DVFS also have limited effectiveness during low utilization periods. As illustrated in Figure 2(a), the processor would be the most energy efficient if the frequency can be scaled so that R1 can run slowly and finish just-in-time to meet the target tail latency. If the frequency cannot scale this low, then significant power savings opportunity can be lost.

This is also demonstrated in Figure 1b. At service times above 250μs, Rubik outperforms DynSleep due to the benefits of DVFS in slowing down requests. However, with shorter request service time, DVFS cannot *extend* request processing time, but Deep Sleep can *delay* request processing time, in order to exploit latency slack for power savings.

Summary: From Observations 3 and 4, the key obstacles to DPM under microsecond request service times are (1) inefficient sleep state management, and (2) DVFS' transition overhead and limited ability in closing the latency gap at low utilization. To solve these issues, this paper presents μDPM, which aggressively deep sleeps to minimize idle periods, delay wakeup to met C-state target residency time, and coordinate frequency scaling to complete the request just-in-time to meet the target tail latency constraint.

Can latency-critical workloads utilize deep sleep states? It has been widely assumed that sleep states are not applicable under strict tail latency constraints [13]. Surprisingly, we found that deep sleep modes can be beneficial as long as it is tail latency aware.

As shown before, short service times in latency-critical workloads cause shorter idle period lengths, which often times hampers sleep states and lead to performance penalties. However, ***the order-of-magnitude difference between the service time and the target tail latency provides significant opportunity for dynamic power management.***

Figure 2(b) illustrates this key insight, utilizing latency and service time measurements from SPECjbb with tail service time of 78μs and tail latency target of 800μs. A request can arrive at any time when the core is in C6 and can be safely delayed and processed without missing the deadline (even in the worst-case scenario of R1 arriving just as the core transitions to C6). As long as the difference between the target tail latency and the tail service time is somewhat greater than the C6 residency time, deep sleep states can be effectively utilized. Compared to Figure 2(a), μDPM also enables the processor to stay in deep sleep longer and to make full use of the latency gap while coordinating with DVFS.

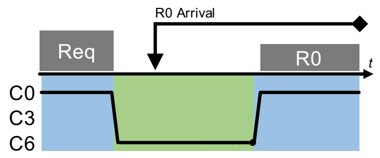
Figure 2(c) illustrates a scenario with even tighter latency constraints (150μs), and extremely short request service time (33μs). In this scenario, the tail latency target is shorter than the C6 residency time. Therefore, many requests that enter during C6 will likely result in QoS violation (except for the requests entering towards the end of the residency time). In this scenario, QoS gets the priority and we will wake the core up earlier with the goal of processing the request just-in-time. Note that compared to Figure 2(a), the amount of wasted energy is less. *Note that naturally existing idle periods do exist, and μDPM can opportunistically consolidate these idle periods with latency slack from request delaying to meet residency time or minimize sleep overheads.* No other DPM technique is able to achieve this.

III. μDPM

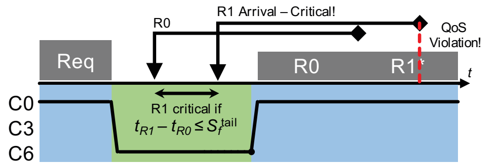
In this section, we detail μDPM, a dynamic power management scheme for the microsecond era. We demonstrate that through *careful coordination of DVFS, sleep, and request delaying, we can overcome the challenges of microsecond-era applications.* In addition, to minimize state transition overheads, we introduce a *criticality-aware scheduler* to coordinate scheduling across multi-core in order to keep cores in its current power state.

A. Overview

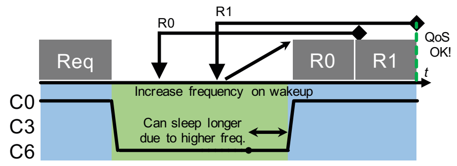
The key insight driving μDPM is that by carefully coordinating DVFS, sleep, and request delaying we can delay and slow down request processing to finish just-in-time to meet the target tail latency, even under the constraints of microsecond request service time.



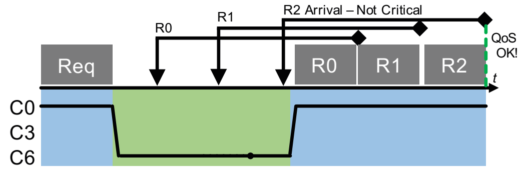
(a) Aggressive deep sleep and request delaying



(b) Critical request arrival


R1
R0
QoS OK!
Req
R0
R1
t
Increase frequency on wakeup
C0
C3
C6
Can sleep longer due to higher freq.

(c) Re-configuration on critical request arrival



(d) Non-critical request arrival

Figure 3: μDPM run-time illustrative example. (a) Aggressively deep sleep when idle, and delay wakeup until the request can finish just-in-time at lowest frequency setting. (b) Arriving R1 is a critical request and will miss latency target. (c) Increase frequency on wakeup to meet R1 latency target. Due to the higher frequency, we can sleep longer as R1 will complete faster. (d) A normative case with non-critical request arrival. Request arrival indicator ▼. Target tail latency indicator ◆.

We illustrate μDPM in Figure 3. In Figure 3(a), the moment a core is idle (e.g., when all prior requests complete), μDPM immediately goes into the deepest sleep state (C6) in order to save idle power. μDPM needs to maintain (1) when to wake up, and (2) what frequency to run upon wake up. We refer to these two parameters as a *configuration*. Upon entering sleep, the configuration is reset with a null wake-up time (representing stay asleep indefinitely), and with the lowest frequency setting.

When R0 arrives, the tail service time of R0 is predicted while running at the current frequency configuration, and a wake-up time is set such that R0 finishes just-in-time. Instead of waking up the core at the latest moment and processing at full speed to meet tail latency targets, we wake up the core earlier and process the request at a slower frequency to achieve a better trade-off between latency and power savings. This is in contrast to Baseline, Rubik, and SleepScale which wakes up upon request arrival; DynSleep which runs at highest frequency; and DreamWeaver which uses static timeout-based request delaying.

Figure 3(b) illustrates a scenario where a second request, R1, arrives during an idle period and is determined to violate QoS constraints given the current wake-up time and frequency configuration. We define requests that will violate QoS constraints as a *critical request*. Since we know that the previous request satisfies QoS constraints, we can simply detect a critical request by comparing if the inter-arrival time between these two most recent requests is less than the predicted tail service time of the incoming request.

Whenever a critical request arrives, we reconfigure the wake-up time and frequency configuration. We first increase the frequency until R1 meets QoS. Unlike Rubik, we only increase frequency, and not decrease, to limit DVFS transition overhead. Since frequency increased, R1 will complete faster enabling μDPM to sleep longer, increasing the idle period length and still satisfy QoS.

Figure 3(c) shows a normative case where another arriving request, R2, is not critical. In this scenario, R2 is satisfied with the given wake-up time and frequency, and will therefore simply queue. Also, if a critical request arrives during an active period, this simply triggers a frequency increase as the wake-up time is void.

μDPM needs to determine: (1) When to wake up after sleeping? and (2) What frequency to run at? The key is to estimate the incoming request's service time. This is especially challenging in data centers due to short request service time, which causes significant short-term variability that often dominates tail latency [11, 38]. To account for this, μDPM utilizes a statistical-based performance model [11] and criticality-aware scheduling to recalculate wake-up time and frequency at every critical request arrival. In addition, μDPM will also consider transition overheads while determining the optimal wake-up time and runtime frequency.

B. Performance Modeling

Estimating Request Tail Service Time: To estimate the tail service time of processing and queued requests, we utilize a statistical performance model based on [11]. This model has been previously shown to be able to highly accurate and can account for the high-variability in latency-critical applications, as well as in capturing uncertainties from co-location and memory interference through online periodic resampling (ever 100ms) of service cycles distribution. With the addition of precomputed target tail tables, this model can also compute the required frequency constraints for each incoming request.

At a high-level, this model breaks down request processing into two probability distributions: *cycles* spent in compute, P[$C = c$], and *time* spent memory-bound, P[$M = t$]. These probability distributions can be sampled online through performance counters, for P[$C = c$], and through *CPI stacks*[11, 39, 40] for P[$M = t$]. Because of the non-deterministic request demands, the service time of a request is often considered as a random variable. Previous work similarly assumes that the service time for each request is drawn independently from a single distribution [5, 11]. While it has been shown previously that different request types can also have different distributions [27], we choose to utilize a single distribution for simplicity, trading off a small amount of power savings opportunity.

When multiple requests are in the queue, it is not sufficient to just estimate request tail service time. *We require estimating the completion time of the requests upon wake up.* Therefore, the estimated completion *cycle* of a request Ri is a random variable S_i, with probability distribution P[$S_i = c$]. The completion cycle distributions all draw from a single distribution P[$S = c$], where S gives how many cycles it takes to process one request. S is essentially a combination of the compute cycle distribution, C, and memory time distribution, M; $S = C + Mf$. To obtain the tail service request time, we draw the 95th percentile of the distributions.

The cycle at which Ri completes, P[$S_i = c$], can then be computed as the n-fold convolution ($*$) of S, where n is the number of queued request and processing request. Unlike [11], we simplify our model by not conditioning the currently processing request on elapsed cycles completed. For example, in Figure 3(d), the estimated completion cycle of R2 (the random variable S_2) is the sum of the random variables S_0, S_1, and S, and is estimated as the following convolution: P[$S_2 = c$] = P[$S_0 = c$] $*$ P[$S_1 = c$] $*$ P[$S = c$]. Completion cycle can be simply converted to completion time by dividing the core's frequency.

Estimating Request Tail Latency: In order to determine whether a request is critical or not, we first need to estimate that latency of a given request. The estimated tail latency of the request, L_i, is given as follows:

$$L_i = W + T_{wake} + T_{dvfs} + \frac{S_i}{f} \tag{1}$$

, where W is the time until the core is scheduled to wake up, T_{wake} is wake up transition time, T_{dvfs} is the DVFS transition time, and S_i is the estimated tail completion cycle to service request Ri as discussed prior, and f is the operating frequency. Based on this latency model, we can relate target tail latency, core frequency, and wake up time to determine μDPMconfigurations.

Determining Critical Requests After estimating the latency of arriving requests, we need to check whether or not the arriving request is critical to determine whether a configuration update is needed. By observing Figure 3 and equation (1), a request is critical if

$$t_{R_i} - t_{R_{i-1}} \leq \frac{S^{tail}}{f} \tag{2}$$

, where t_R is the arrival time of a request R, and S is the completion cycle distribution for a single request (service only, no queueing). This is because the arrival time between the current and previous request is too short for processing one request. Intuitively, for a given wake-up/VFS configuration, when the processor wakes up, it can process a request every $\frac{S_i}{f}$ seconds and meet the target tail latency just in time. If requests arrive too close together, since the previous request is scheduled to finish just in time to meet the target tail latency, the current request will experience longer latency than the previous one, exceeding the tail latency target (as illustrated in Figure 3(b)). We will now leverage this insight to simplify the calculation for new wake-up time and frequency configurations.

Determining New Wake-Up Time and Frequency Configuration. A critical request occurs when the incoming request cannot meet QoS requirements. Therefore, we take a 2-step approach. First, we need to determine the new frequency. Conceptually, this can be illustrated in Figure 3(b) as, What is the frequency requires to squeeze R1 to fit between R0 and the red line? This can be achieved as

$$T_{R_i TargetCompletion} - T_{R_{i-1} Completion} = \frac{S^{tail}}{f'} \tag{3}$$

, where $T_{R_i TargetCompletion}$ is the target tail latency completion time of R*i*, and $T_{R_{i-1} Completion}$ is the completion time of R*i-1*, S is the completion cycle distribution for a single request, and f' is the new frequency. Since all of these variables are available by the time a request is determined to be critical, the new frequency can be computed directly. To minimize DVFS transition overheads, we limit frequency changes to only increase, and not decrease. The frequency would then reset to a lower level upon the next idle period.

Next, we determine the wake-up time. If a critical request arrives during an active period, then this step is not necessary. As illustrated in Figure 3(b), this can be achieved by essentially shifting all requests to the right, or as late as possible, while still satisfying latency constraints. This requires re-estimation of completion cycles for all queued requests. [11] observed that queue size are typically under 10, and can be quickly re-sampled using target tail tables. To determine the new wake-up time, we only need to get the new completion time for the first queued request, S_0, and then utilize Equation 1 to compute the new wake-up time. If the wake-up time is determined to be shorter than the residency time, then μDPM will wake up at the cost of some energy overhead, as illustrated in Figure 2(c).

Impact of Mispredictions: A misprediction on request service time is equivalent to misidentifying a critical request and potentially violating SLA. Since our target tail latency is based on 95th percentile *tail latency*, 5% of requests are allowed to finish slower without violating SLA. Similarly, we estimate 95th percentile *tail service time*, where there is a 5% chance of misprediction for the first delayed request. The chance of SLA violation decreases significantly for subsequent queued requests due to summing the random variables S_i. For example, S_1's estimate is that the two queued requests (R0 and R1) will both finish at the tail, which is extremely rare (0.25% chance). Therefore, our estimates are already very conservative.

C. Minimizing State Transition Overheads

As shown in Figure 1d, transition overheads can account for significant portions of power consumption. In μDPM, we incur low power state transition overhead whenever a critical request triggers configuration updates. While effective at meeting the tail latency target, these configuration updates result in wasted power. It is because the core will sleep less and do no useful work during the state transition. In this section, we propose a *criticality-aware configuration scheme* to redirect requests and avoid unnecessary configuration updates. In addition, this also helps absorb bursty request spikes. Algorithm 1 shows the pseudo-code for this scheme.

When a request arrives at a core, there are 4 possible states that the request could face based on its criticality and core's status: (1) the request is critical and the core is sleeping (*critical-sleeping*); (2) the request is critical and the core is active (*critical-active*); (3) the request is not critical and the core is sleeping (*noncritical-sleeping*) and (4) the request is not critical and the core is active (*noncritical-active*).

When a request is either noncritical-sleeping or noncritical-active, no configuration update is needed, thus no additional state transition will occur. A noncritical-sleeping core is a core that is sleeping and already scheduled to wake up so that the request will meet tail latency target. When a request is critical-sleeping, the wake-up time and/or frequency need to be updated. No additional C-state or DVFS transition is introduced because the core is sleeping and only the frequency after wakeup and/or the scheduled time of wakeup are changed.

The only case where a new transition is introduced is when the core is sleeping with no scheduled wakeup (the core is sleeping and have no arrived requests), and when a request is critical-active where an additional DVFS transition is needed. The goal of the criticality-aware scheduler is to redirect the request to the core which makes it non-critical. By doing so, we decrease the chances of triggering configuration updates and DVFS state changes.

Based on equation (2), we define the criticality score of a request for a core as:

$$criticality = \frac{S^{tail}/f}{t_{R_i} - t_{R_{i-1}}} \tag{4}$$

, hence non-critical request will have criticality score less than 1. Our algorithm first computes the criticality score of that request for each core, and if there exists only one core which makes the request non-critical, that core is selected for processing. However, it is more than likely that multiple cores can make the incoming request non-critical. In this case, the non-critical sleeping cores are chosen for the request to avoid the possibilities for future DVFS transition. If multiple cores are still candidates, we select the core which will process a request with the lowest *extra energy*.

The energy consumption for a given configuration (wake-up time W and frequency f) is:

$$E(W,f) = \left(W - T_{sleep}\right) P_{idle} + \left(T_{sleep} + T_{wake}\right) P_{max} + T_{dvfs} P_{dvfs} + \left(\frac{S_i}{f}\right) P_f \quad (5)$$

The first term in equation (5) is the idle period energy with the idle power P_{idle} and sleep transition time T_{sleep}. The second term is the transition energy overhead of going into and out of sleep mode. The third term is the DVFS transition energy, and the last term is the active period energy consumption with the estimated completion time of the last queued request, S_i/f, multiplied by the active power P_f. Note that we assume the idle period started at $t = 0$. If a core candidate is already active, then the transition terms will be 0. On the other hand, there are some cases that no core can make the request non-critical, then we favor mapping the request to the core where it initially arrived.

Another benefit brought from our criticality-aware scheduler is to mitigate bursty requests. When a burst of requests arrives, especially during prolonged sleep periods, most of the requests will be identified as critical. Our scheduler will attempt to schedule them to cores that make them "non-critical", resolving burstiness. From algorithm 1, we can see that the overhead of the criticality-aware scheduler is linear to the number of cores in the processor. In our experiment, this overhead is less than 2 μs and is considered as part of the request service time.

μDPM Implementation: μDPM simply requires the performance models detailed previously. In order to derive and calculate service time distributions, we make use of precomputed target tail tables [11]. Prior work has shown that these tables can be periodically updated every 100ms with only 0.2% overhead. Since we assume per-core DVFS and per-core sleep state, we require a μDPM implementation within every core. Due to the frequency of resampling service time distribution, target tail tables are load sensative and are able to capture high-variability unknowns such as interference effects from co-location or multi-core interference on shared resources. In addition, these target tail tables can be incorporated into processor's PCU to calculate wake-up time and frequency state directly.

Algorithm 1: Criticality-aware scheduling

```
1:  $non\_critical\_cores = \phi, non\_critical\_sleep\_cores = \phi$
2:  for each core do
3:      compute $core_i$'s criticality
4:      if $criticality \leq 1$ then
5:          $non\_critical\_cores \leftarrow non\_critical\_cores \cup core_i$
6:          if $core_i$ is sleeping then
7:              $non\_critical\_sleep\_cores \leftarrow$
                $non\_critical\_sleep\_cores \cup core_i$
8:          end if
9:      end if
10: end for
11: if $non\_critical\_cores \neq \phi$ then
12:     if $non\_critical\_sleep\_cores \neq \phi$ then
13:         return $min(extra\ energy)$ in $non\_critical\_sleep\_cores$
14:     else
15:         return $min(extra\ energy)$ in $non\_critical\_cores$
16:     end if
17: else
18:     return $min(extra\ energy)$ in $all\ cores$
19: end if
```

To detect critical request, we showed previously that we only need to keep track of the previous incoming request's timestamp and the service time of the new request. In addition, to calculate wake-up time in Equation (1) and frequency in Equation (3) only requires simple arithmetic computations once the service time has been determined by the precomputed target tail tables. To support request redirection for criticality-aware scheduling, we require either software-level support through the request handling stack (network, or asynchronous event libraries such as libevent), or through hardware support with SmartNICs, where the scheduling is offloaded to the NIC interface.

IV. EVALUATION

We evaluate μDPM using an in-house simulator with various latency-critical data center workloads representative of common microservices. Our simulator is a framework for stochastic discrete-time simulation of a generalized system driven by empirical profiles of a target workload (based on BigHouse [30]). Empirical inter-arrival and service distributions are collected from measurements of real systems at fine time-granularity. Using these distributions, synthetic arrival/service traces are generated and fed to a discrete-event simulation that models the server's active and idle low-power modes. Latency measures (e.g., tail response latency) are obtained by logging the start and finish time of each request. Similarly, energy measures are obtained through the weighted sum of the duration of idle, busy and transition periods with their corresponding power consumption. Similar to previous works [11, 13, 14, 24, 41], we focus on CPU power which is the single largest contributor to server power.

The power consumption at each processor C-state and frequency step is collected from measurements of real systems and is shown in Table I. Also, the uncore power (LLC and some peripheral circuit) is measured as 10W. We model a 12-core server, similar to our experimental server. Our processor can support frequency from 1.2GHz to 2.7GHz. At peak, our processor consumes 45W overall and 18W at active idle. Unless otherwise stated, we use 10 μs as DVFS transition time, respectively. As mentioned previously, we use a 10μs DVFS transition time as many prior works have empirically observed DVFS transition times between 6-70μs [33–37], even with fast on-chip integrated voltage regulators.

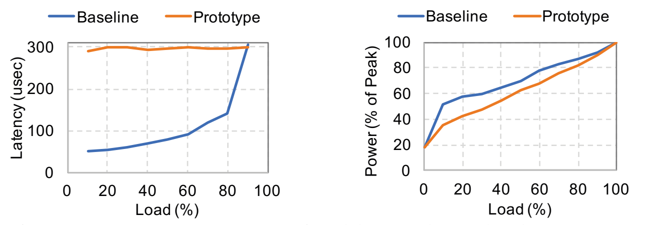


Figure 4: Prototype μDPM is able to close the latency gap and achieve lower power compared to Baseline.

Deep sleep states (C6) flushes and power gates local L1 and L2 caches, and can incur longer transitions into sleep and performance overheads due cold misses on wake up [42]. We use 89 μs as sleep state transition time, and conservatively double this value when entering sleep to account for cache flushing. To model cold miss penalty, we add an additional 25μs to the service time of the first request that arrives during idle [42]. We also conservatively estimate maximum power usage during sleep transition events.

A. *Proof-of-concept Validation*

We constructed a proof-of-concept, implemented with a combination of user-space and kernel-space modifications, to demonstrate the feasibility μDPM. The server has dual Intel Xeon-E5 12-core processors, with 136GB memory. In our experiments, Hyperthreading and Turbo Boost are disabled. The details of each sleep state and frequency were provided previously. The Baseline configuration uses the default Linux Menu idle governor and DVFS handled by intel_pstate driver.

μDPM consists of aggressive deep sleep, request delaying, and coordinated DVFS. To implement aggressive deep sleep we modified the Linux idle driver to go straight into C6. Due to a lack of per-core DVFS in our processor, we set the frequency at maximum. Therefore, the results presented in our prototype is conservative. Request delaying is handled through libevent, an asynchronous event handling library.

For our prototype, we implement μDPM with Memcached [43, 44], which we observed to have the shortest service time and shortest tail latency target (Table 2). We use two multicore servers, one client and one request processing server. The client server establishes multiple TCP connections with the server, emulating multiple clients. The inter-request time distribution for each client is exponential and is scaled to see the impact of different traffic loads.

We run Memcached [43] across 12 threads, with a fixed one-to-one thread-to-core mapping to avoid run-to-run variance caused by the Linux thread scheduler [45]. We separate the original worker thread into two threads: a scheduler thread and request processing thread. The scheduler thread consists of the libevent wrapper and the criticality-aware scheduler, to also help queue and direct requests. In the request processing thread, we run the μDPM performance models to monitor the queues and determine wake up time and frequency configurations. The request processing thread

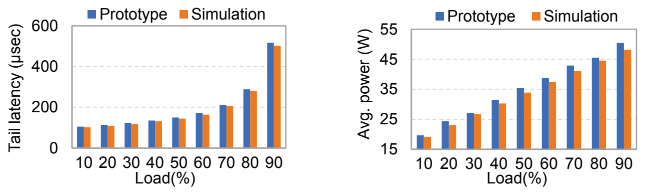


Figure 5: Prototype vs simulation validation.

has the ability to directly handle DVFS by directly writing to MSRs. All scheduler threads are running on one core mapped to the second processor.

Figure 4 presents the result of our proof-of-concept. μDPMis highly effective at closing the latency slack, even with the target tail latency as short as 150 μs and without the use of DVFS. Furthermore, at low loads where idleness exists, μDPM is able to save a significant amount of power using only sleep states without violating sub-millisecond tail latency constraints. At low utilization, μDPM can reduce power consumption by 18% of the peak power, with an almost linear energy proportionality profile.

Simulator Validation: In addition, we validated our simulation infrastructure against our prototype as shown in Figure 5. In this figure, we ran Memcached with our prototype server with Hyperthreading and Turboboost disabled. We ran Memcached across a range of loads in order to verify that our simulator's power and performance model agrees with a real system. The latency figure, which plots the tail latency of the Baseline machine configuration, and the power figure clearly demonstrates that our simulation model agrees well with the actual behavior of the prototype system.

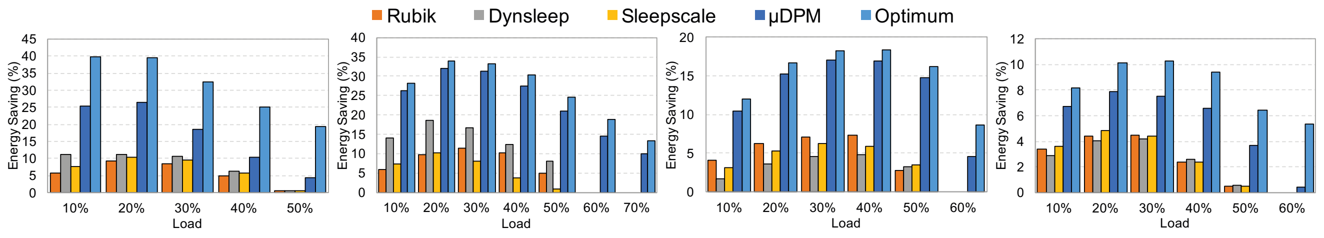
B. *Workloads*

Typical web application may consist of front-end web server (e.g. Apache, NGINX), business logic (e.g. accounts, catalog, inventory, ordering), databases or data stores (e.g. Memcached, MongoDB, Redis), or specialized functionalities, such as search engines and recommendation systems. We select four different latency-critical workloads that are representative of typical backend microservices.

Table II gives the workload characteristics of four applications we use in our evaluation. Throughout this paper, we define the tail latency at the 95th percentile and the target tail latency (SLA) as the tail latency of the applications running at medium load without applying any power management (similar to [11]). The service time was discussed in depth in section II. We later show the result of varying target tail latency and the effect on power savings in section IV-C.

Table II: Workload characteristics.

Name	Avg. Service Time	Tail Service Time	Target Tail Latency
Memcached [43, 44]	30μs	33μs	150μs
SPECjbb [8, 11]	65μs	78μs	800μs
Masstree [8, 11, 46]	246μs	250μs	1100μs
Xapian [8, 11]	431μs	1200μs	2100μs



(a) Memcached (b) SPECjbb (c) Masstree (d) Xapian

Figure 6: Energy saving comparisons among different power management schemes.

Memcached and **Masstree**: We evaluate a common in-memory key-value store used in many production data centers, such as Google, Facebook and Twitter. Due to the simple computational requirements, key-value stores involve significantly less processing time per request (tens of microseconds). Because of this, key-value store has even tighter tail latency constraints, typically on the order of hundreds of microseconds. Key-value store adds to our analysis an example of an ultra-low latency workload. We captured traces (request arrivals and service time) from a real server running *Memcached*. In addition, we use a different service time distribution from *Masstree* [46] provided in [8, 11].

SPECjbb: Business logic makes up a large component of many large-scale web applications such as e-commerce. These business logic can include various functionality such as sales, inventory, and customer management. We evaluate SPECjbb [47], a Java middleware benchmark that simulates supermarket company with various microservices covering the services of supermarkets, suppliers, and headquarters.

Search: We evaluate the query serving portion of a production web search service. Search requires thousands of leaf nodes all running in parallel in order to meet the stringent tail latency constraints. In search, most of the processing is in the leaf nodes and account for the vast majority of nodes in a search cluster, and are responsible for an overwhelming fraction of power consumed[5, 48, 49]. We use the service distributions of *Xapian*, provided in [8, 11] as a representative search algorithm.

C. Results

We compare our results with three state-of-the-art power management schemes: Rubik [11] (VFS-only scheme), DynSleep [24] (sleep-only schemes), and SleepScale [22] (sleep+VFS scheme). The results of Rubik are representative of other DVFS-only schemes, such as Pegasus and TimeTrader [13, 23]. We model an optimistic Rubik design that sets the maximum frequency as the nominal frequency. SleepScale combines both Sleep state and DVFS selection. SleepScale requires long computation time for its simulation-based prediction scheme, and therefore calculates Sleep and DVFS selection after every 1 second epoch [22]. For DynSleep we set the frequency at maximum and always enters C6 deep sleep state upon encountering an idle period. DynSleep is representative of other Deep Sleep techniques, such as PowerNap [6] and DreamWeaver [18], but outperforms both due to its ability to harness per-core level idleness instead of socket-level (full-system) idleness. In addition, we also show the results for the "Optimum" scheme which has no transition overheads and operates as follows: the core will enter the deepest sleep state when idle and wake up instantly as soon as the request arrives, similar to DynSleep. During request processing, DVFS configuration is adjusted at each request arrival instance, similar to Rubik. The idealized optimum scheme serves as an upper bound of energy saving.

We consider processor level energy consumption, which includes both core and uncore power. For energy saving results, all schemes are compared with the default baseline scheme, where all CPU cores are operated under maximum frequency, and uses the Linux menu governor [32] to choose the C-state during the idle periods. We also use the Linux menu governor for C-state selection for Rubik.

Energy Savings: In figure 1d, we gave the energy consumption breakdowns and show that μDPM can effectively reduce busy and idle power, and the state transition power. μDPM only incurs a 3% energy overhead for transition. Here, we give the sensitivity studies of the energy saving on different traffic loads and applications. Figure 6 reports the energy saving results for four different applications across different CPU loads. The CPU loads are adjusted by scaling the inter-arrival time of requests.

Across all workloads, μDPM provides significantly more energy savings. Typically, μDPM is within 2-3% of the Optimum scheme, except for Memcached, which exhibits the shortest service times. Here, transition time overheads begin to dominate and μDPM starts to fall behind Optimum. Nevertheless, μDPM still provides ~2x energy savings compared to all other schemes.

Another noteworthy trend is that at high utilization, most schemes are not able to provide any power savings at all because the utilization exceeds the utilization that the target tail latency was set. However, in some cases, like in SPECjbb, Masstree and Xapian, μDPM is still able to squeeze out modest energy savings where all other DPM fails. These results demonstrate that request delaying and sleep states are the key to saving power at high utilization with sub-millisecond latency constraints.

For extremely low latency workloads, such as *Memcached* and *SPECjbb*, the major energy inefficiency is the idle period energy consumption as demonstrated earlier in this paper. Therefore, sleep-based techniques provide better energy savings than DVFS-based techniques. SleepScale achieves good energy saving by selecting the best C-state and DVFS pair through design space exploration. However, because of the short natural idle periods, cores can only enter shallow sleep states during the short idle periods. Therefore, significant energy saving can be achieved by delaying request processing to create longer idle periods, as in the case of DynSleep. With μDPM, we can achieve up to 26% energy saving for *Memcached* and 32% for *SPECjbb*, compared with the next best technique, DynSleep, at 12% and 18%, respectively.

For applications with relatively higher request processing time (*Masstree* and *Xapian*), the achieved energy savings are relatively lower as the baseline scheme is also able to take advantage of longer natural idle periods and enter deep sleep states. All existing schemes perform similar in terms of energy saving. DynSleep provides only moderate energy saving. Longer processing time means longer nature idleness, and further prolonging these idle periods will only give marginal return in energy saving. SleepScale typically outperforms all other schemes, except for μDPM, because it chooses the C-state and DVFS configuration with lowest power by exploring the entire C-state and DVFS's design space. Longer idle periods here allows SleepScale to naturally enter deeper sleep states. μDPM achieve up to 17% energy saving for *Masstree* and 8% for *Xapian*, compared with the next best technique, at 8% and 5%, respectively.

Contrary to existing assumptions that sleep states cannot be used in latency-critical workloads [11, 13, 23], *these results demonstrate the need to coordinate delay request processing, sleep and DVFS power management with microsecond latency constraints.*

Reducing State Transition Overhead: Figure 7 shows the normalized state transition count with our criticality-awareness algorithm. The results are normalized to the sum of C-state and DVFS transition counts in μDPM without criticality-awareness scheduling. We can see that criticality-awareness effectively reduces the C-state and DVFS transition overhead to less than 20% of the baseline μDPM. For all applications, at low load, the majority overhead reduction is for DVFS, and at high load, for C-state. At low load, it is easier to find the sleeping core which makes requests non-critical. As a result, more DVFS transition is avoided. On the other hand, it is harder to find cores that can make requests non-critical at high load, hence criticality-awareness will try to avoid C-state transition since it has higher power overheads.

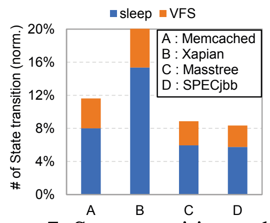


Figure 7: State transition reduction with criticality-awareness.

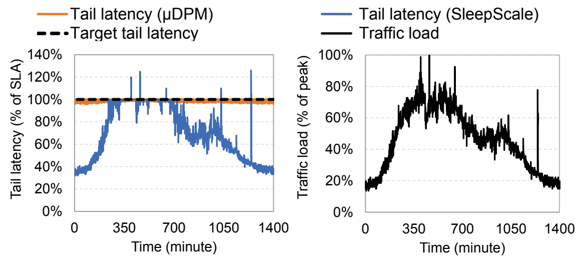
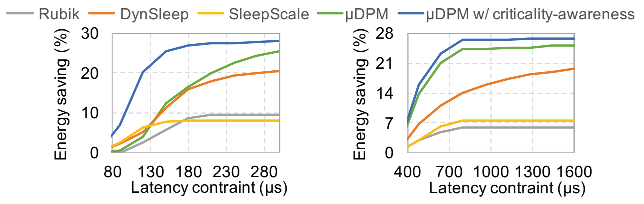
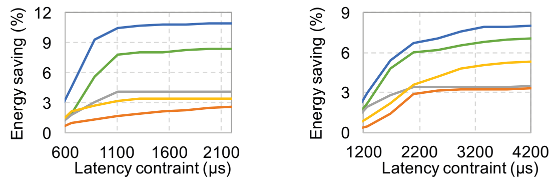


Figure 8: Tail latency under varying traffic load.



(a) Memcached (b) SPECjbb



(c) Masstree (d) Xapian

Figure 9: Sensitivity to target tail latency.

Responsiveness to Load Changes: μDPM's design also allows it to respond instantaneously to sudden changes in input load. Since wake-up time and frequency are updated at every critical request arrival, μDPM reacts to the changes in the inter-request arrival time immediately (Rubik and DynSleep have similar behavior). Figure 8 shows how μDPM responds to sudden load change with *Masstree*. We compare with SleepScale using a 60s epoch as in [22]. The input load trace is shown on the right, which we gather from our institution's data center. The target tail latency is defined as the tail latency under maximum load (80%) from the trace. The 95th percentile latency for SleepScale and μDPM, sampled every 1 second, is shown in the left. First, we can see that SleepScale cannot close the latency gap under low load and cannot always satisfy the target tail latency; due to its history-based prediction, it can not react to sudden load changes, resulting in multiple target tail latency violations. μDPM is able to achieve stable tail latencies under all loads. At low load, μDPM delay requests and wake up the core later, processing requests at a lower rate, and achieve tail latency close to the latency bound while saving additional power. Also, it adapts quickly to sudden load changes and burst, through criticality-aware scheduling which can absorb bursts through scheduling to cores with least criticality.

Sensitivity to Tail Latency Constraint: In data centers, the target tail latency is usually defined at the peak load. Previously, we assumed that the SLA is set at 50% load. However, there might be some cases that the peak load is

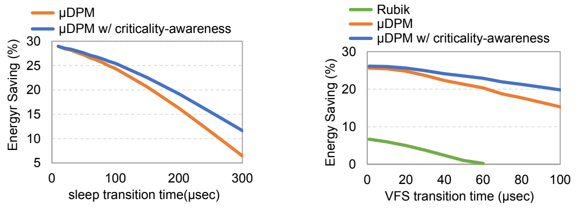


Figure 10: Sensitivity to transition time.

different. Figure 9 shows the energy savings with different tail latency constraints. The x-axis is the latency constraint for each application, and the y-axis is the energy savings achieved for that target tail latency constraint. As expected, energy savings generally decreases as the SLA gets tighter in all four applications and all schemes. The rate of this decrease is highly related to the application characteristics. For Memcached, the energy saving quickly drops to 0 as latency constraint decreases under all schemes, except μDPM with criticality-awareness because the latency slacks become smaller than the resident time of deep sleep state and the DVFS transition latency. We achieve better energy saving as latency constraint decreases due to the coordination request delaying and request rescheduling among cores. For other workloads, the energy savings are more resistant to the latency constraint because of the relatively larger latency slack. μDPM consistently outperforms all other schemes in four applications and all latency constraint levels. *μDPM consistently outperforms all other schemes by saving power over 3x in Masstree, and 2x in SPECjbb and Xapian.*

V. DISCUSSION AND CONCLUSION

Need for faster DVFS and sleep transition. The challenges presented by microsecond service times also motivates the need for faster low power state transitions to take advantage of shorter idle periods and shorter service times. For example, faster DVFS transitions would require improvements to integrated voltage/frequency regulators [33], and faster sleep states can be achieved through integration of non-volatile memory components [50, 51].

Even with the introduction of faster low power state transition, careful coordination of DPM is still required. In Figure 10, we characterize how improved DVFS and sleep transition impacts the effectiveness of μDPM. We present results for *Memcached* at 10% load. For comparison, we present the results for Rubik, in Figure 10 (right). Even with idealized 0μs transition time, μDPM outperforms Rubik due to the added benefits of coordinating deep sleep and request delaying. Since μDPM determines the configuration considering both sleep and DVFS transition overheads, it will dynamically balance the use of each scheme. With higher state transition overheads, criticality-awareness sees clear benefits by reducing state changes. This enables μDPM to be more resistant to state transition overhead changes.

Opportunities from microservice chains. For microservice applications, end-to-end latency typically involves a chain of individual microservices [52]. There holds significant promise in coordinating latency slack for power savings across microservice chains [52–54]. We hope μDPM can form the foundation for future work that borrow latency slack from other microservices.

Attack of the Killer Microseconds. With the emergence of microservice software architectures and low-latency I/O devices (e.g. NVM storage, faster data center networking), the "killer microsecond" presents challenges across the entire data center stack. Recent works have tackled the challenges of thread scheduling for microsecond applications [55, 56] and hiding microsecond device access latency [3]. Concurrent to our work, Duplexity [57] seeks to improve server utilization and efficiency in the face microsecond-scale stalls. Our work hopes to contribute to the foundation of this emerging problem space by highlighting the challenges and opportunity for dynamic power management in the "killer microsecond" era.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their invaluable comments and suggestions. This work is partly supported by NSF grant CCF-1815643, California Energy Commission grant EPC-16-030, and the University of California, Riverside.

References

[1] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, 2014.

[2] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, pp. 48–54, Mar. 2017.

[3] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the killer microsecond," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[4] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[5] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," *IEEE micro*, vol. 23, no. 2, pp. 22–28, 2003.

[6] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," in *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[7] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[8] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict qos for latency-critical workloads," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[9] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *International Symposium on Microarchitecture (MICRO)*, 2011.

[10] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[11] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *International Symposium on Microarchitecture (MICRO)*, 2015.

[12] D. Wong and M. Annavaram, “Knightshift: Scaling the energy proportionality wall through server-level heterogeneity,” in *International Symposium on Microarchitecture (MICRO)*, 2012.

[13] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, “Towards energy proportionality for large-scale latency-critical workloads,” in *Intl. Symposium on Computer Architecuture (ISCA)*, 2014.

[14] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, “Tradeoffs between power management and tail latency in warehouse-scale applications,” in *International Symposium on Workload Characterization (IISWC)*, 2014.

[15] D. Wong and M. Annavaram, “Implications of high energy proportional servers on cluster-wide energy proportionality,” in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2014.

[16] D. Wong, “Peak efficiency aware scheduling for highly energy proportional servers,” in *International Symposium on Computer Architecture (ISCA)*, 2016.

[17] E. Le Sueur and G. Heiser, “Dynamic voltage and frequency scaling: The laws of diminishing returns,” in *HotPower*, 2010.

[18] D. Meisner and T. F. Wenisch, “Dreamweaver: Architectural support for deep sleep,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[19] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, “Leakage current: Moore’s law meets static power,” *IEEE computer*, vol. 36, no. 12, 2003.

[20] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, “Microarchitectural techniques for power gating of execution units,” in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[21] A. Lungu, P. Bose, A. Buyuktosunoglu, and D. J. Sorin, “Dynamic power gating with quality guarantees,” in *International Symposium on Low Power Electronics and Design (ISLPED)*, 2009.

[22] Y. Liu, S. C. Draper, and N. S. Kim, “Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers,” in *International Symposium on Computer Architecuture (ISCA)*, 2014.

[23] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar, “Timetrader: Exploiting latency tail to save datacenter energy for online search,” in *Intl. Symposium on Microarchitecture (MICRO)*, 2015.

[24] C.-H. Chou, D. Wong, and L. N. Bhuyan, “Dynsleep: Fine-grained power management for a latency-critical data center application,” in *Intl. Symp. on Low Power Electronics and Design (ISLPED)*, 2016.

[25] 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*, vol. PP, no. 99, pp. 1–1, 2016.

[26] D. H. K. Kim, C. Imes, and H. Hoffmann, “Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics,” in *International Conference on Cyber-Physical Systems, Networks, and Applications (ICCPS)*, 2015.

[27] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. G. Dreslinski, “Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting,” in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[28] C. H. Chou and L. N. Bhuyan, “A multicore vacation scheme for thermal-aware packet processing,” in *International Conference on Computer Design (ICCD)*, 2015.

[29] M. Elnozahy, M. Kistler, and R. Rajamony, “Energy conservation policies for web servers,” in *USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[30] D. Meisner, J. Wu, and T. F. Wenisch, “Bighouse: A simulation infrastructure for data center systems,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[31] “Intel idle driver for linux,” http://lxr.free-electrons.com/source/drivers/idle/intel_idle.c.

[32] V. Pallipadi, S. Li, and A. Belay, “cpuidle: Do nothing, efficiently,” in *Proceedings of the Linux Symposium*, 2007.

[33] E. A. Burton, G. Schrom, F. Paillet, J. Douglas, W. J. Lambert, K. Radhakrishnan, and M. J. Hill, “Fivrfully integrated voltage regulators on 4th generation intel® core socs,” in *Applied Power Electronics Conference and Exposition (APEC)*, 2014.

[34] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice, “The TURBO diaries: Application-controlled frequency scaling explained,” in *USENIX Annual Technical Conference*, 2014.

[35] D. Hackenberg, R. Schne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, “An energy efficiency feature survey of the intel haswell processor,” in *International Parallel and Distributed Processing Symposium Workshop*, 2015.

[36] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby, “Evaluation of cpu frequency transition latency,” *Computer Science - Research and Development*, vol. 29, no. 3, pp. 187–195, Aug 2014.

[37] Y. Bai, V. W. Lee, and E. Ipek, “Voltage regulator efficiency aware power management,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[38] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, “Adaptive parallelism for web search,” in *European Conference on Computer Systems (EuroSys)*, 2013.

[39] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, “Predicting performance impact of dvfs for realistic memory systems,” in *International Symposium on Microarchitecture (MICRO)*, 2012.

[40] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, “A performance counter architecture for computing accurate cpi components,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[41] X. Fan, W.-D. Weber, and L. A. Barroso, “Power provisioning for a warehouse-sized computer,” in *International Symposium on Computer Architecture (ISCA)*, 2007.

[42] M. Arora, S. Manne, I. Paul, N. Jayasena, and D. M. Tullsen, “Understanding idle behavior and power gating mechanisms in the context of modern benchmarks on cpu-gpu integrated systems,” in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[43] “Memcached,” http://memcached.org/.

[44] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the clouds: A study of emerging scale-out workloads on modern hardware,” in *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[45] 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 *Symposium on Cloud Computing (SoCC)*, 2014.

[46] Y. Mao, E. Kohler, and R. T. Morris, “Cache craftiness for fast multicore key-value storage,” in *European Conference on Computer Systems (EuroSys)*, 2012.

[47] “Specjbb,” http://www.spec.org/jbb2013/.

[48] L. A. Barroso and U. Hölzle, “The case for energy-proportional computing,” *IEEE Computer*, 2007.

[49] J. Dean, “Challenges in building large-scale information retrieval systems: Invited talk,” in *International Conference on Web Search and Data Mining (WSDM)*, 2009.

[50] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M. Chiang, Y. Yan, B. Sai, and H. Yang, “A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops,” in *Proceedings of the ESSCIRC*, 2012.

[51] X. Pan and R. Teodorescu, “Nvsleep: Using non-volatile memory to enable fast sleep/wakeup of idle cores,” in *International Conference on Computer Design (ICCD)*, 2014.

[52] Y. Hu, C. Li, L. Liu, and T. Li, “Hope: Enabling efficient service orchestration in software-defined data centers,” in *International Conference on Supercomputing (ICS)*, 2016.

[53] C. Delimitrou, “The hardware & software implications of microservices and how big data can help,” in *Eighth Workshop on Architectures and Systems For Big Data (ASBD)*, 2018.

[54] H. Yang, Q. Chen, M. Riaz, Z. Luan, L. Tang, and J. Mars, “Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained CMP,” in *International Symposium on Computer Architecture (ISCA)*, 2017.

[55] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, “Arachne: Core-aware thread management,” in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[56] A. Sriraman and T. F. Wenisch, “µtune: Auto-tuned threading for OLDI microservices,” in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[57] A. Mirhosseini, A. Sriraman and T. F. Wenisch, “Enhancing Server Efficiency in the Face of Killer Microseconds,” in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2019.