NMAP: Power Management Based on Network Packet Processing Mode Transition for Latency-Critical Workloads

Ki-Dong Kang, Gyeongseo Park, Hyosang Kim, Mohammad Alian[†], Nam Sung Kim[‡], Daehoon Kim
DGIST, [†]University of Kansas, [‡]University of Illinois at Urbana-Champaign
{kd_kang, gspark, hyosangkim, dkim}@dgist.ac.kr, alian@ku.edu, nskim@illinois.edu

ABSTRACT

Processor power management exploiting Dynamic Voltage and Frequency Scaling (DVFS) plays a crucial role in improving the data-center's energy efficiency. However, we observe that current power management policies in Linux (i.e., governors) often considerably increase tail response time (i.e., violate a given Service Level Objective (SLO)) and energy consumption of latency-critical applications. Furthermore, the previously proposed SLO-aware power management policies oversimplify network request processing and ignore the fact that network requests arrive at the application layer in bursts. Considering the complex interplay between the OS and network devices, we propose a power management framework exploiting network packet processing mode transitions in the OS to quickly react to the processing demands from the received network requests. Our proposed power management framework tracks the transitions between polling and interrupt in the network software stack to detect excessive packet processing on the cores and immediately react to the load changes by updating the voltage and frequency (V/F) states. Our experimental results show that our framework does not violate SLO and reduces energy consumption by up to 35.7% and 14.8% compared to Linux governors and state-of-the-art SLO-aware power management techniques, respectively.

CCS CONCEPTS

• **Hardware** → **Enterprise level and data centers power issues**; • **Computer systems organization** → *Client-server architectures.*

KEYWORDS

Data-center server, Power management, Dynamic voltage and frequency scaling, Tail latency

ACM Reference Format:

Ki-Dong Kang, Gyeongseo Park, Hyosang Kim, Mohammad Alian, Nam Sung Kim, and Daehoon Kim. 2021. NMAP: Power Management Based on Network Packet Processing Mode Transition for Latency-Critical Workloads. In *Proceedings of The 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3466752.3480098

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO '21, October 18–22, 2021, Virtual Event, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480098

1 INTRODUCTION

It is essential for data-center servers to provide fast response time (i.e., low tail latency) for clients along with high energy efficiency [10, 11]. Since a processor consumes a significant portion of a server's electricity, power management policies (i.e., governor) that adjust voltage and frequency states (i.e., V/F states or Performance states (P states)) [27] as well as sleep states (i.e., C states) of processor cores play a crucial role in improving the energy efficiency.

However, current dynamic power management governors based on CPU utilization, such as `ondemand` governor [37], often increase the tail latency (e.g., 99^{th} percentile (P99) latency) of network requests, and consequently fail to satisfy Service Level Objectives (SLOs) especially when bursts of network packets are received at a server [10]. This is because the default CPU utilization based governors in Linux (e.g., `ondemand` governor in `cpufreq` driver, `powersave` governor in `intel_pstate` driver) cannot quickly change the P state of a core to a state that satisfies the processing demand of the packet bursts. On the other hand, statically operating the core at the highest performance state (i.e., P0 state) is a performance overkill and results in high power consumption and energy waste.

Prior studies propose SLO-aware DVFS policies to improve energy efficiency while satisfying the SLO of latency-critical applications. We can classify these prior studies into long-term DVFS [7, 19, 24–26, 30, 34, 35] and short-term DVFS [1, 8, 16, 21, 41] depending on the scaling period of V/F states. Long-term policies adjust the processor's V/F states every several hundred milliseconds. Such V/F state adjustments are based on the feedback from the application and state of OS. The Long-term policies can offer energy efficiency for latency-critical applications with non-bursty traffic or applications without tight SLO requirements. But when processing a rapidly changing load (e.g., bursty traffic), the V/F state setting is delayed, which often causes SLO violations or energy inefficiency by setting processor's V/F to a high state [21]. Since these studies often rely on complex models for determining the next V/F states that often require tens of milliseconds for inference, it is not feasible to decrease the decision making period of such studies [34, 35]. Meanwhile, short-term DVFS studies [8, 16, 21, 41] provide high energy efficiency by quickly changing the V/F state in response to the load changes. However, the proposed techniques rely on special voltage regulators and assume that the V/F state transition time is negligible (e.g., even several tens of nanoseconds [16]). Moreover, these studies feed their power management model with synthetic load and ignore the overheads of network software stack processing as well as the complex interplay between OS and network interface card (NIC).

Tackling limitations of prior power management techniques, we propose `NMAP`, **N**etwork packet processing **M**ode-**A**ware **P**ower

management which is a short-term DVFS technique readily applicable to the commercial processors. NMAP considers packet processing status on each core by monitoring transitions between network packet processing modes – interrupt and polling supported by Linux New API (NAPI) [39]. NMAP leverages the processing mode transitions to quickly detect the bursts in network traffic and react to the bursts in a timely manner by adjusting the processor's V/F states. This is to improve energy efficiency while preventing SLO violation. More specifically, when the number of packets processed in the polling mode increases, it indicates that a burst of packets is received (and continued to be received) at the NIC that requires the processor to constantly poll the NIC. NMAP increases the V/F state when excessive packet processing occurs in the polling mode to prevent SLO violations. On the other hand, when the number of packets processed in interrupt mode increases, NMAP sets the processor's V/F states according to the V/F state decision of a CPU utilization based governor to reduce the unnecessary energy consumption. Although polling events can happen in short successions, NMAP only uses these events to increase V/F state at the burst's early part and to fall back to the CPU utilization based governor, which uses a sufficiently long period for decision making (e.g., 10ms), at the burst's end. Consequently, NMAP does not require repetitive V/F state transitions in a very short period of time while it can react to sudden bursts in a timely manner, which enables NMAP to be readily deployed for commodity processors.

We first introduce the simplified NMAP that implements a power management policy that only considers the events of ksoftirqd in polling mode. Since ksoftirqd is woken up when the packet processing in the softirq handler is delayed, NMAP can simply exploit the scheduling information of ksoftirqd without any prior information from applications. Next, we propose NMAP that considers the ratio of packets processed in interrupt and polling modes for making power management decisions.

In addition to proposing NMAP, we discuss experimental analysis of V/F state transition latency and the impact of sleep states on the response latency of latency-critical applications. We first demonstrate that repetitive V/F state transitions lead to additional transition latency (i.e., *re-transition latency*), making it challenging to implement short-term DVFS studies on current commercial processors. Our experimental analysis shows that the *re-transition latency* increases to hundreds of microseconds depending on the processor. Such high *re-transition latency* prevents quick V/F state updates and in turn can lead to request's latency increase and SLO violations. Furthermore, we demonstrate that the impact of sleep states on the tail response latency of latency-critical workloads is negligible. We observe that the measured wake-up latency from the deepest sleep state is tens of microseconds, which does not hurt tail latency of the latency-critical application in a millisecond scale. Consequently, the energy efficiency can be improved by constantly enforcing the deepest sleep state when the core is idle.

For evaluation, we implement NMAP on a real system equipped with Intel Xeon processor supporting per-core DVFS and use two representative latency-critical applications, memcached [14], an in-memory key-value store, and nginx [38], a lightweight web server. Our experimental results show that NMAP reduces energy consumption by up to 35.7% and 14.8% compared to conventional Linux governors and state-of-the-art SLO-aware power manager [1], respectively. Our results show that the end-to-end response time of the applications is always within the SLO when NMAP is used for power management.

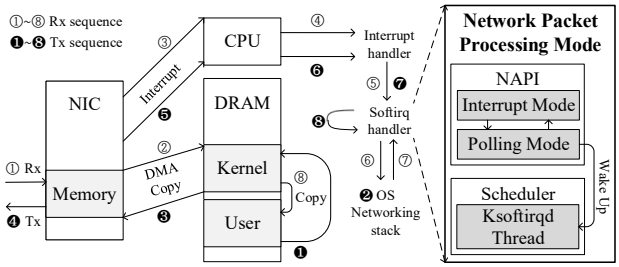


Figure 1: Network packet processing with NAPI.

2 BACKGROUND

2.1 Network Packet Processing and New API (NAPI)

New API (NAPI) [39] is an interface implemented in Linux by default to mitigate the number of interrupts when receiving or transmitting network packets (i.e., Rx or Tx) through the polling mechanism. While processing the network packets, NAPI transitions between interrupt and polling modes based on packet processing status and network loads. Fig. 1 illustrates Rx and Tx sequences with NAPI. NIC notifies a core of packet arrivals (③) or transmission completion (❺) using an interrupt. Subsequently, the interrupt handler processes the interrupt (④ and ❻) and invokes softirq handler that processes a packet (⑤ and ❼) or repeats to process packets waiting in Rx and Tx queues (⑦ and ❽) in the interrupt or polling mode, respectively. For Rx, the softirq handler delivers packets to the OS networking stack by extracting packets from the Rx buffer, whereas the softirq handler for Tx checks the completion of Tx and cleans Tx buffers used if Tx is correctly completed. In the polling mode, softirq handler disables the interrupt for network I/O, it can process packets without additional interrupt delivery and handler invocation.

However, if the kernel continuously processes packets in polling mode as the network I/O occurs frequently, application threads can be rarely scheduled on the core since the softirq handler has the higher scheduling priority than the application threads. Therefore, the Linux kernel provides ksoftirqd that has the same scheduling priority with application threads to prevent the starvation of application threads [33]. When the system is booted, the Linux kernel creates a ksoftirqd thread for each core. The softirq handler migrates the remaining packet processing to ksoftirqd thread in the following conditions: 1) the softirq handler overuses schedule ticks more than two ticks for processing packets (e.g., 8ms in 250Hz configuration), 2) the softirq handler fails to empty Rx and Tx queues more than a certain number of iterations (e.g., more than ten iterations), 3) the softirq handler yields the current core to process scheduler when reschedule flag is set by events (e.g., Inter-Processor Interrupt (IPI)). In terms of network I/O handling, ksoftirqd is thus invoked when the processing of Tx/Rx packets continuously occurs.

Such NAPI-based packet processing is performed by cores that receive/transmit network packets. Since conventional or some of

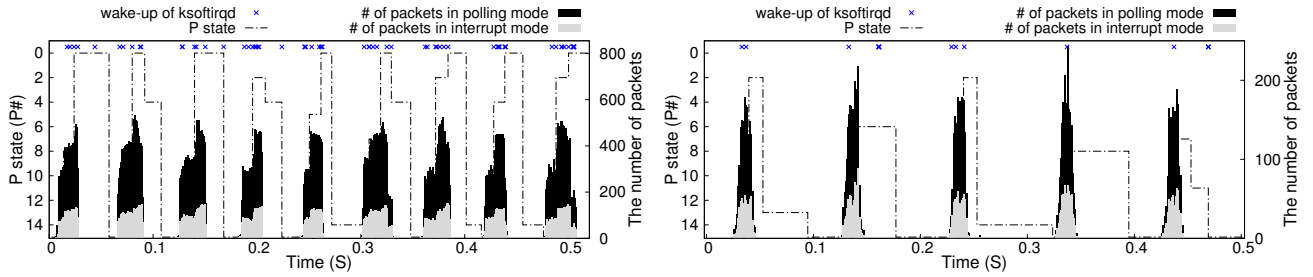


Figure 2: Wake-up of ksoftirqd, P state by the ondemand governor, the number of packets processed in interrupt mode and polling mode.

recent NICs receive/transmit packets using a single core, only single core performs the NAPI-based packet processing. However, some other recent NICs support multiple queues that enable multiple cores to receive/transmit the packets for the high performance network. With the multi-queue NICs, each core processes packets separately while transitioning between interrupt and polling mode, and invoking ksoftirqd based on its status.

2.2 Processor Power Management

V/F State Management. Commercial processors support DVFS to dynamically change processor's performance states (i.e., P states) based on the speculated future load on the processor. Linux OS provides several power management policies (i.e., governors) to determine processor's P state with cpufreq driver. The cpufreq driver includes the performance, powersave, userspace, ondemand, and conservative [6] governors. The performance and powersave governors set and maintain the maximum and minimum V/F state of the processor, respectively, while the userspace governor sets and maintains the V/F state specified by the user. The ondemand and conservative governors dynamically set the V/F state based on the CPU utilization that is measured periodically (e.g., every 10ms). While the ondemand governor determines the V/F state based on CPU utilization only, the conservative governor gradually adjusts the next V/F state by transitioning to a value near the current V/F state (e.g., P1→P0 or P1→P2).

Current Intel processors support intel_pstate driver for power management including the performance and powersave governors[1]. The intel_performance governor operates cores at the maximum V/F state similar to the performance governor in the cpufreq driver. The intel_powersave governor determines the next V/F state based on the CPU utilization similar to the ondemand governor of cpufreq driver. The intel_powersave governor also provides optimizations for improving storage I/O performance. The optimizations include increasing V/F state when a block I/O request completes or a new thread is dispatched to a core.

Modern server-class processors support per-core DVFS where each core can deploy its own governor and independently set its V/F state. Processors that do not support per-core DVFS often support chip-wide or per-cluster DVFS. In such processors, multiple cores share the same V/F state even though the governor on each core determines a different V/F state for each core. The V/F state of processors supporting chip/cluster DVFS is set to the highest V/F state among the V/F states determined by the governor deployed on each core.

Sleep State Management. Commercial processors support sleep states that turn off some components of the processor, such as register, cache, and memory controller, to reduce power consumption. In modern processors, each core has multiple sleep states, called Core C states (CC) (e.g., CC0, CC1, CC6). In CC0 (or active state), the processor is active and executes instructions. In CC1 (or clock gated state), the processor is halted and is not clocked. CC6 (or deep idle state) power off the core, registers, and private caches (i.e., L1 and L2 caches). The deeper the core C state, the lesser the energy consumption at the cost of a higher performance penalty for waking up and restarting execution. To reduce power consumption without notable performance degradation, Linux deploys menu governor [36] by default. The menu governor predicts the future idle time of a core – based on the core's utilization history – and sets the core's C state accordingly. The processor stays at a C state until the predicted sleep time by menu governor expires or an external event such as an interrupt wakes up the core.

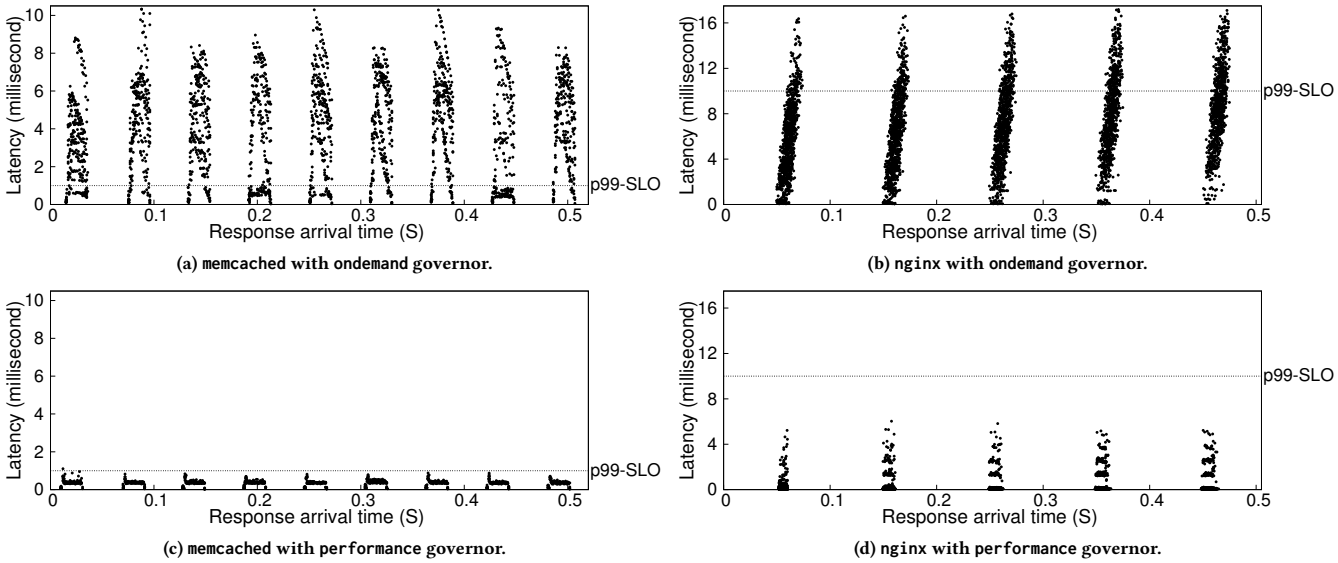
3 MOTIVATION

3.1 Network Packet Processing Mode Transition of NAPI and Network Load

NAPI processes Rx or Tx packets in either interrupt or polling mode based on the status of Rx and Tx queues. Such mode transitions are directly affected by the network load and the packet processing performance. We run memcached and nginx server to quantitatively demonstrate the correlation between mode transitions of NAPI and network load. We send 750K and 56K requests per second (RPS) from 20 client threads running on a separate physical machine to the server. The client generates repetitive bursts of network packets along with idle periods. We use 750K and 56K RPS for memcached and nginx as a high load throughout this paper.

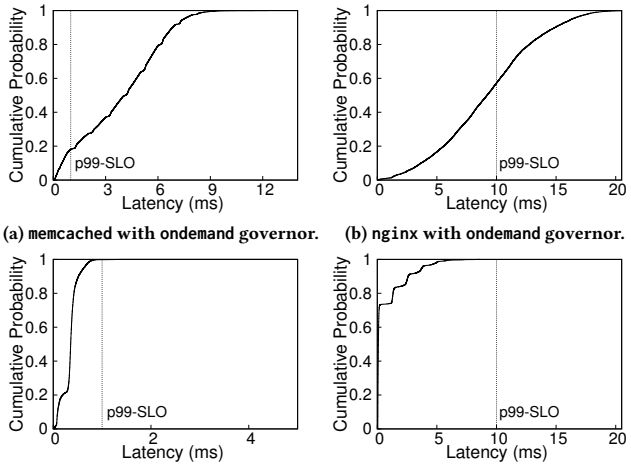
The right y-axis in Fig. 2(a) and Fig. 2(b) show the number of network packets received (sampled every 1ms) on memcached and nginx servers, respectively. The left y-axis in the figures shows the changes in the P state of the core that runs one of the memcached or nginx threads. We use ondemand governor for the experiments. We split the number of packets into two stacked bars, each of which represents the portion of network packets processed in interrupt or polling modes. We also mark the times when ksoftirqd wakes up. At each burst, as the load goes to the peak, the polling rate increases proportionally to the load; however, the number of packets processed in the interrupt mode is capped. In our experiment, the number of packets processed in interrupt mode does not exceed

[1]To prevent confusion, we refer to the governors of intel_pstate drivers as intel_performance and intel_powersave from now on in the paper.



(a) memcached with ondemand governor.
(b) nginx with ondemand governor.
(c) memcached with performance governor.
(d) nginx with performance governor.

Figure 3: Response latency of every packets during 0.5 second.



(a) memcached with ondemand governor.
(b) nginx with ondemand governor.
(c) memcached with performance governor.
(d) nginx with performance governor.

Figure 4: Cumulative distribution function (CDF) of response latency.

152 and 89 for memcached and nginx, respectively. This behavior is expected as at a high network Rx rate, the NIC keeps filling the Rx queue and the processor repetitively processes the pending packets in the polling mode and frequently wakes ksoftirqd up. On the contrary, as the load decreases, the core empties queues more often, leading to termination of softirq and transition to interrupt mode. NMAP leverages this intuitive observation and piggybacks the existing NAPI mechanism to quickly react to the load changes on a server running latency-critical applications (Sec. 4.2).

3.2 Limitation of CPU utilization based Power Management

CPU utilization based governors such as ondemand and intel_powersave often fail to react to the bursts of network packets [1, 25] in a timely manner, leading to SLO violations. The culprit in these governors is that the sampling period for CPU utilization and decision making is orders of magnitudes larger than the lifetime of an Rx packet burst (10ms vs. 100s of μs). This mismatch results in either not detecting short-lived bursts or late reaction to longer bursts.

As plotted in Fig. 2, the ondemand governor mostly raises the V/F state in the middle or later part of the packet bursts. Another key observation from Fig. 2 is that ondemand governor does not immediately set the processor's P state to P0, even when it detects an Rx burst. This means that even when ksoftirqd is woken up, the processor is not running at the maximum V/F while the softirq handler requires fast processing of incoming packets.

Fig. 3 plots the end-to-end response time for each memcached and nginx requests sent over a 0.5 second interval, respectively. The x-axis in the figures represents the time when a response is received at the client-side. Fig. 4 plots the cumulative distribution function (CDF) of response latency in the environment shown in Fig. 3. We set the SLO for the applications to the inflection point of the latency-load curve as prior studies do [25], which is 1ms and 10ms for memcached and nginx, respectively. As plotted in Fig. 3(a) and Fig. 3(b), the ondemand governor fails to satisfy the SLO. Since the SLO is defined as P99 response time, it mandates that 99% of the requests should have a lower response time than the SLO. Fig. 4(a) and Fig. 4(b) show that only 18.1% and 57.2% of the requests have a shorter response time than 1ms and 10ms for memcached and nginx, respectively. To validate that the improper V/F states are the main culprit for SLO violations, we run the same experiments with the performance governor. As shown in Fig. 3(c) and Fig. 3(d), a server deploying performance governor significantly reduces response time of each request during a burst. As plotted in Fig. 4(c) and Fig. 4(d), performance governor satisfies the SLO while only 0.14% requests have a longer response time than 1ms for memcached and all requests are processed within 10ms for nginx, i.e., P99 is less than the target SLO.

4 NMAP: NETWORK PACKET PROCESSING MODE AWARE POWER MANAGEMENT

In this paper, we propose NMAP, **N**etwork packet processing **M**ode-**A**ware **P**ower management. The key idea is to monitor network packet processing mode transitions in NAPI and decide on the future V/F state of the processor in a per-core basis. NMAP implements a

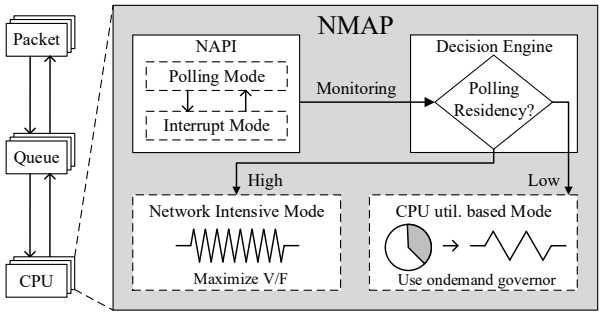


Figure 5: NMAP architecture.

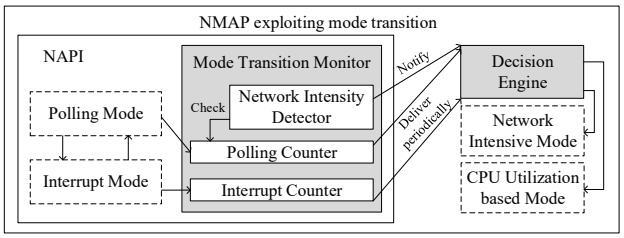


Figure 6: NMAP exploiting network packet processing mode transitions.

"proactive" power management policy as it leverages the packet processing delay in the NAPI for early adjustment of the processor's V/F states. When NMAP detects excessive packet processing on cores based on NAPI's mode, it maximizes the V/F state to process packets faster. Otherwise, if NMAP detects that the cores are processing incoming packets fast enough, it falls back to the default CPU utilization based governor.

4.1 NMAP Architecture

Fig. 5 shows the overall architecture of NMAP. Based on the packet processing status in NAPI, `Decision Engine` determines an appropriate power management mode between `Network Intensive Mode` and `CPU Utilization based Mode`. NMAP chooses the `Network Intensive Mode`, which operates cores at the highest V/F state (i.e., P0), when it detects that excessive packet processing occurs on each core to satisfy the SLO of latency-critical applications. Otherwise, NMAP falls back to the `CPU Utilization based Mode` that adjusts the V/F state based on the current CPU utilization. NMAP targets multi-core processors supporting per-core DVFS, which processes packets in parallel on each core with multi-queue NICs. NMAP monitors the packet processing status in NAPI and determines power management mode for each core.

We first introduce the simplified version of NMAP that exploits the scheduling information of `ksoftirqd` to determine when it changes the power management mode. As discussed in Section 2.1, `ksoftirqd` is woken up when the softirq handler processing network packets in the polling mode overuses the scheduler ticks or fails to empty Tx/Rx queues repetitively. This means that we can deduce that the core fails to process packets fast enough along with excessive packet processing when `ksoftirqd` is woken up. Therefore, NMAP promotes the `Network Intensive Mode` that maximizes the V/F state when `ksoftirqd` is woken up. On the contrary, when `ksoftirqd` falls into sleep after it completes packet processing, NMAP falls back to the CPU utilization based governor (e.g., ondemand governor) to consume power based on the loads on the core.

Algorithm 1: Mode Transition Monitor

```
1:  NI_TH = threshold for Network Intensive Mode
2:  pkt_poll = packets processed in polling mode
3:  pkt_intr = packets processed in interrupt mode
4:  if pkt_poll > NI_TH then
5:      send a notification to Decision Engine
6:  end if
7:  poll_cnt = poll_cnt + pkt_poll
8:  intr_cnt = intr_cnt + pkt_intr
9:  if periodic timer is expired then
10:     send poll_cnt, intr_cnt to Decision Engine
11:     poll_cnt = intr_cnt = 0
12: end if
```

Algorithm 2: Decision Engine

```
1:  CU_TH = threshold for CPU Util. based Mode
2:  if receive a notification from monitor then
3:      set Network Intensive Mode
4:      disable ondemand governor
5:      maximize current V/F state
6:  else
7:      if is Network Intensive Mode then
8:          if ratio of polling to interrupt < CU_TH then
9:              set CPU Util. based Mode
10:             enforce P state based on CPU util.
11:             enable ondemand governor
12:         end if
13:     end if
14: end if
```

NMAP based on `ksoftirqd` is simple and readily applicable with any workloads while not requiring profiling/monitoring running applications' behaviors. However, raising the V/F state when `ksoftirqd` is woken up may be not fast enough to satisfy SLOs in particular for high loads. This necessitates a more sophisticated technique that can react to the network bursts under high load adequately.

4.2 NMAP Exploiting Network Packet Processing Mode Transition

In this section, we propose NMAP exploiting transitions of network packet processing mode of NAPI between interrupt and polling. NMAP observes the current status of packet processing on each core by monitoring the number of packets processed in interrupt mode or polling mode. Then, NMAP calculates the ratio of the number of packets processed in the polling mode to the number of packets processed in interrupt mode to detect that excessive packet processing occurs on the core; note that the ratio of polling to interrupt increases as the network load increase as discussed in Section 3.1.

Fig. 6 illustrates the overall architecture of NMAP exploiting network packet processing mode transitions, specifically polling to interrupt ratio. NMAP consists of two components, `Mode Transition Monitor` and `Decision Engine`. `Mode Transition Monitor` tracks mode transitions between interrupt and polling, and notifies `Decision Engine` of the packet processing status of each core. Based on the packet processing status delivered by the monitor, `Decision Engine` chooses a power management mode between `Network Intensive Mode` and `CPU Utilization based Mode`.

Algorithm 1 describes how the `Mode Transition Monitor` operates to observe packet processing status. The monitor maintains two counters, each of which counts the number of network packets processed in polling and interrupt mode, respectively (line 2-3). `Network Intensity Detector` examines network intensity by checking the counter for polling; the increase in the polling ratio

means the increase in the number of pending packets. Therefore, if the number of packets processed in the polling mode exceeds a specified threshold, the monitor sends `Decision Engine` a notification that a core cannot process packets fast enough with the current V/F and it needs to raise the V/F of the core (line 4-6). The monitor accumulates packets processed in interrupt and polling modes (line 7-8) and also delivers the counted values to `Decision Engine` and resets the counters to zero (line 9-12) periodically.

Algorithm 2 describes how `Decision Engine` chooses a power management mode. `Decision Engine` executes the algorithm periodically or when it receives a notification from the monitor. `Decision Engine` chooses the `Network Intensive Mode` when it receives the notification. For the `Network Intensive Mode`, `Decision Engine` disables the governor (i.e., ondemand governor) and maximizes current V/F (line 4-5). On the contrary, `Decision Engine` calculates the ratio of polling to interrupt periodically and chooses the `CPU Utilization based Mode` when the ratio is smaller than a specified threshold (line 8-12). Since the decrease in polling ratio means that there are fewer packets pending in queues since a core processes the packets fast enough. For the `CPU Utilization based Mode`, `Decision Engine` enforces a CPU utilization based P state and re-enables the ondemand governor (line 10-11). Since the ondemand governor usually sets the V/F state to one lower than P0 as discussed in Section 3.2, `NMAP` can reduce power consumption. `NMAP` does not require complicated and time-consuming profiling of applications while requiring the simple threshold for the power management mode transition.

As described in algorithms for `Mode Transition Monitor` and `Decision Engine`, `NMAP` uses two thresholds (i.e., *NI_TH* and *CU_TH*) for the mode transition between `Network Intensive Mode` and `CPU Utilization based Mode`. `NMAP` needs to reset the threshold values when the running application changes, but it does not need to reset the values when the running application's load changes. `NMAP` finds a threshold value that can set to `Network Intensive Mode` at the early part of the network burst before the load reaches to the peak of the burst. `NMAP` also finds another value that can fall back to `CPU Intensive Mode` when the load decreases from the peak of the burst to a certain point where the core can process the packets fast enough.

Specifically, `NMAP` obtains the threshold values via off-lined, but light-weight profiling. First, `NMAP` chooses a specific load used to set the target SLO (e.g., inflection point of latency-load curve). At the chosen load, `NMAP` counts the number of packets processed in the polling mode per interrupt at the early part of a request burst; in our implementation, `NMAP` observes the first 100 interrupts from the start of a request burst. Then, `NMAP` uses the maximum value among the measured ones as *NI_TH*. For *CU_TH*, `NMAP` calculates the average polling to interrupt ratio during a single request burst and uses the average ratio as *CU_TH*.

Although `NMAP` obtains the threshold values by monitoring the polling behavior of a single request burst at the specific load instead of the exhaustive search, it requires resetting the values via the profiling for running another application. For dynamic adjustment of the thresholds when the running application changes, `NMAP` resets the thresholds with the specific load (e.g., inflection point of latency-load curve) before the application actually runs. We leave further exploration of on-line profiling techniques as our future work.

Table 1: Re-transition latency (10,000 experiments).

Processor	P state transition	Mean (µs)	Stdev (µs)
Intel i7-6700	$\mathbf{P}_{max} \rightarrow \mathbf{P}_{max-1}$	21.0	2.2
	$\mathbf{P}_{max-1} \rightarrow \mathbf{P}_{max}$	34.6	2.2
	$\mathbf{P}_{max} \rightarrow \mathbf{P}_{min}$	27.2	5.5
	$\mathbf{P}_{min} \rightarrow \mathbf{P}_{max}$	45.1	6.5
	$\mathbf{P}_{min+1} \rightarrow \mathbf{P}_{min}$	25.3	1.4
	$\mathbf{P}_{min} \rightarrow \mathbf{P}_{min+1}$	35.8	2.2
Intel i7-7700	$\mathbf{P}_{max} \rightarrow \mathbf{P}_{max-1}$	21.7	3.8
	$\mathbf{P}_{max-1} \rightarrow \mathbf{P}_{max}$	31.3	2.1
	$\mathbf{P}_{max} \rightarrow \mathbf{P}_{min}$	25.9	3.1
	$\mathbf{P}_{min} \rightarrow \mathbf{P}_{max}$	50.7	6.6
	$\mathbf{P}_{min+1} \rightarrow \mathbf{P}_{min}$	26.3	2.9
	$\mathbf{P}_{min} \rightarrow \mathbf{P}_{min+1}$	33.8	2.3
Intel Xeon E5-2620v4	$\mathbf{P}_{max} \rightarrow \mathbf{P}_{max-1}$	516.1	3.4
	$\mathbf{P}_{max-1} \rightarrow \mathbf{P}_{max}$	516.2	3.5
	$\mathbf{P}_{max} \rightarrow \mathbf{P}_{min}$	520.9	5.6
	$\mathbf{P}_{min} \rightarrow \mathbf{P}_{max}$	520.3	5.9
	$\mathbf{P}_{min+1} \rightarrow \mathbf{P}_{min}$	517.2	4.3
	$\mathbf{P}_{min} \rightarrow \mathbf{P}_{min+1}$	517.2	4.2
Intel Xeon Gold 6134	$\mathbf{P}_{max} \rightarrow \mathbf{P}_{max-1}$	525.7	5.7
	$\mathbf{P}_{max-1} \rightarrow \mathbf{P}_{max}$	525.6	5.7
	$\mathbf{P}_{max} \rightarrow \mathbf{P}_{min}$	528.4	7.0
	$\mathbf{P}_{min} \rightarrow \mathbf{P}_{max}$	527.3	7.1
	$\mathbf{P}_{min+1} \rightarrow \mathbf{P}_{min}$	526.3	6.4
	$\mathbf{P}_{min} \rightarrow \mathbf{P}_{min+1}$	526.9	6.8

5 DISCUSSION

5.1 Limitation of power management requiring a very short latency of V/F state transition

This section investigates the limitations of power management that requires to change the V/F state in μs scale. According to sub-tables of ACPI (Advanced Configuration and Power Interface) including DSDT (Differentiated System Description Table) and SSDT (System Service Descriptor Table), the V/F transition latency of modern processors (e.g., Intel Xeon and i7 processors) is 10μs. Prior DVFS studies [8, 16, 21, 41] for latency-critical applications require a very short latency when changing the V/F state (as low as several tens of nanoseconds [16]). To set the appropriate V/F state for each request, the latency of V/F state transitions for those studies should be at least equal to or shorter than the request's inter-arrival times. In case when network requests arrive continuously, as modern NICs often deliver network packets to the CPU using interrupts, the inter-arrival time of requests is equal to or shorter than the minimum interrupt generation period. In other words, to adjust the V/F state of each request level, the processor must support the V/F state transition latency shorter than the interrupt generation period of the NIC. For example, with the Intel 82599 NIC [17], the transition latency of the V/F state should be at least equal to or less than 10μs since the interrupt generation period of the NIC is 10μs.

However, when we set a particular V/F state and then transition to another V/F state immediately, it requires longer latency than specified in the ACPI tables as the transition latency of the V/F state (i.e., 10μs). This limitation makes it challenging to apply the power management policies that require a very short V/F transitioning

Table 2: Wake-up time (100 experiments)

Processor	C state transition	Mean (µs)	Stdev (µs)
Intel i7-6700	CC6→CC0	27.70	3.00
	CC1→CC0	0.35	0.48
Intel i7-7700	CC6→CC0	27.56	4.15
	CC1→CC0	0.40	0.49
Intel Xeon E5-2620v4	CC6→CC0	27.25	4.77
	CC1→CC0	0.50	0.50
Intel Xeon Gold 6134	CC6→CC0	27.43	4.05
	CC1→CC0	0.56	0.50

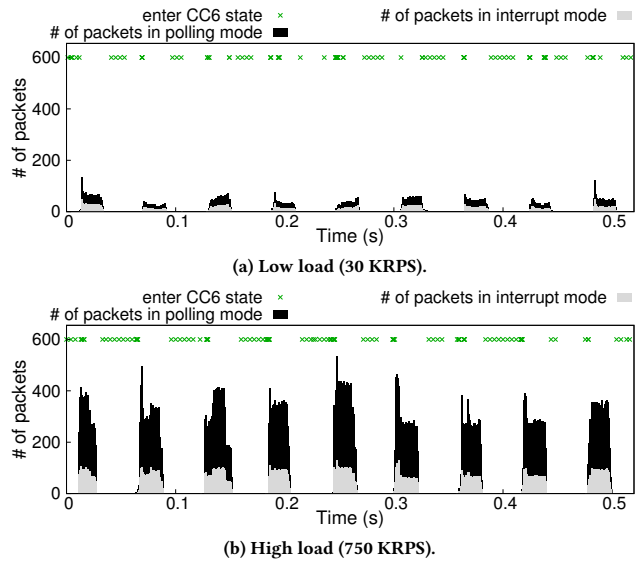
time to current systems. For example, if we change the V/F state from P0 to P1, and then change the V/F state to P2 immediately, it requires longer latency than the normal transition latency. We denote this latency as *re-transition latency* throughout this paper.

For experimental analysis, we measure the *re-transition latency* of two desktop processors (i.e., Intel i7-6700 and i7-7700) and two server processors (i.e., Intel Xeon E5-2620v4 and Gold 6134). We attempt to change the current V/F state by updating the `ctrl` register repetitively, then measure the time until the update is actually reflected. Table 1 shows the average and standard deviation of the *re-transition latency*; we repeat the experiments 10,000 times. As shown in Table 1, the *re-transition latency* of desktop processors is much longer than 10μs, which is specified in the sub-tables of ACPI, by about $2times$–$5times$. In addition, raising the V/F state, especially to the state with a significant difference in the V/F from the current state, requires a longer latency. For example, transitioning $\mathbf{P}_{min}$ to $\mathbf{P}_{max}$ (e.g., P15→P0) requires a longer latency than transitioning $\mathbf{P}_{max-1}$ to $\mathbf{P}_{max}$ (i.e., P1→P0). The *re-transition latency* becomes worse in the cases of server processors (i.e., Intel Xeon). The server processors show about 500μs *re-transition latency* (50× longer than 10μs specified in the ACPI's tables) for all cases while not showing a notable difference between the cases. Such long re-transition latency may make those processors not reflect a considerable number of V/F state transitions performed by prior studies requiring a very short latency for the V/F transition.

5.2 Impact of Sleep State on Latency-Critical Workloads

Although the sleep state (i.e., C state) can considerably contribute to the power reduction of processors, many prior studies [8, 9, 24, 29] mention that the latency-critical applications require attention when using the sleep states because of the wake-up overhead. To investigate the impact of the sleep state experimentally, we measure P99 latency of `memcached` server with `menu` governor, which is the default sleep state governor in Linux. For the experiments, we establish a client server architecture using 10GbE links. To eliminate the impact of the V/F state on the latency, we use the `performance` governor that statically sets V/F to the maximum state.

Fig. 7 shows the number of packets processed in interrupt and polling mode in the server for low (30KRPS) and high (750KRPS) loads. Fig. 7 also plots when the processor enters the deepest sleep state (i.e., CC6) by `menu` governor. As plotted, the processor often enters the deepest sleep state when the processor does not process packets or at the early stage of the burst, whereas it does not enter the deepest state from the middle of the bursts where the CPU



(a) Low load (30 KRPS).

(b) High load (750 KRPS).

Figure 7: Entering CC6 (deepest sleep) state, the number of packets processed in interrupt and polling mode (`memcached`).

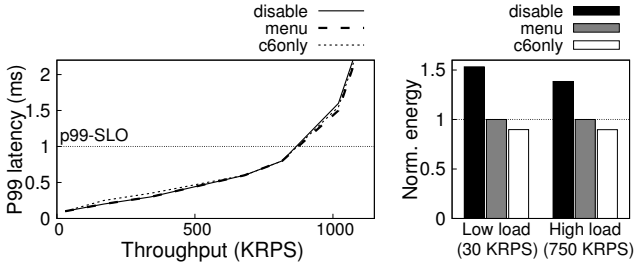
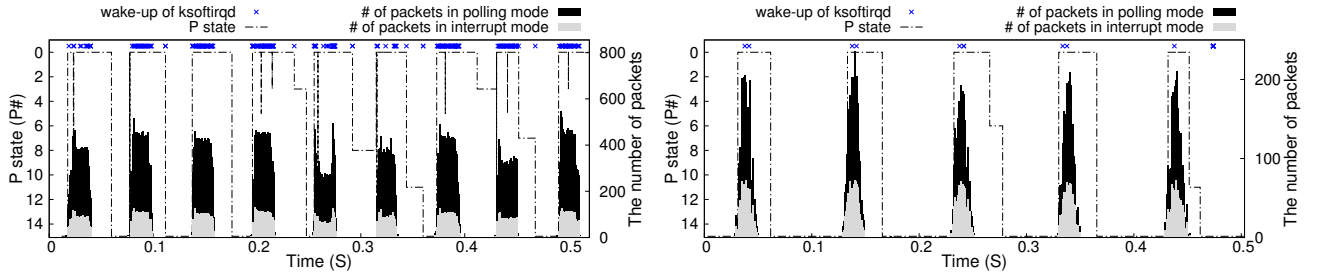


Figure 8: Latency-load curve and energy consumption as the sleep state policy changes. `performance` governor is used for P state and SLO is 1ms.

processes packets intensively. This means that the deepest sleep state, which shows the longest wake-up latency, does not degrade the performance from the middle of the bursts.

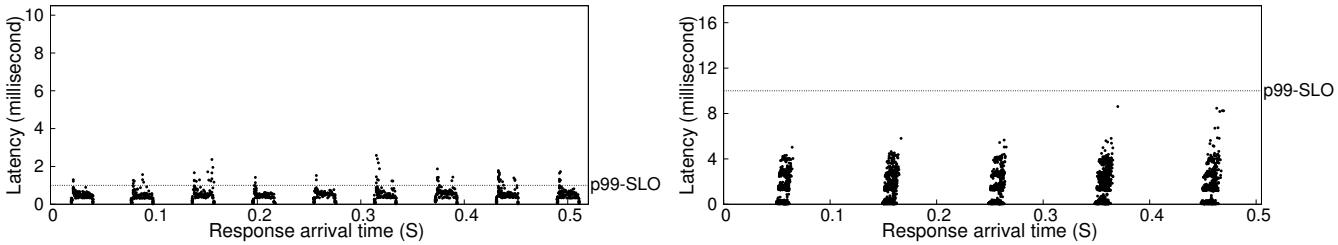
Fig. 8 shows the P99 latency and energy consumption with three different sleep state policies, `menu`, `disable`, and `c6only`, respectively. We disable the sleep state so that the processor never enter the sleep state for `disable`, and enforce the deepest sleep state (i.e., CC6) when the processor falls into the sleep state for `c6only`. The energy results are normalized to `menu`'s results. As shown in the left of Fig. 8, there is no notable difference in P99 latency among the sleep state policies. However, as shown in the right of Fig. 8, compared with `menu`, `disable` consumes more energy by 53.2% while `c6only` consumes less energy by 10.3%. This means that the sleep state policies do not have a notable impact on the tail latency while leading to considerable difference in the energy consumption.

To figure out another reason of the sleep state's negligible impact on the tail latency, we measure the wake-up latency of the sleep states. To measure the wake-up latency of each sleep state, we run a sleep thread and a wake-up thread, each of which runs on a separate core. The wake-up thread sends a signal to the sleep thread that puts a core into a sleep state, and measure the wake-up latency of the core. Table 2 represents the average and standard deviation of the wake-up latency from CC1 to CC0 (i.e., active state) and from



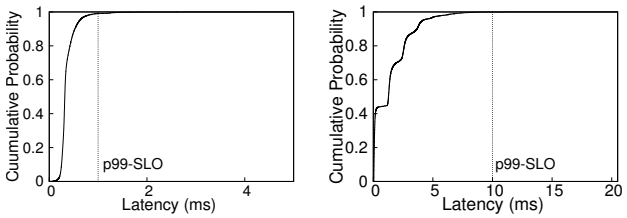
(a) memcached. (b) nginx.

Figure 9: Wake-up of ksoftirqd, P state by NMAP, the number of packets processed in interrupt and polling mode.



(a) memcached. (b) nginx.

Figure 10: Response latency of every packets during 0.5 second with NMAP.



(a) memcached with NMAP. (b) nginx with NMAP.

Figure 11: Cumulative distribution function (CDF) of response latency.

CC6 to CC0. The wake-up latency from the deepest state (i.e., CC6) is about 27μs.

Since CC6 state flushes all cache-lines in the private caches, therefore, we also need to take the effect of private cache flushes into account when measuring the overhead of waking up from CC6 state. To figure out the cost of cache flushing, we measure the time it takes to access all flushed cache-lines after the core is woken up from CC6. We design a micro-benchmark that fills the private caches, and then waits for a certain amount of time to let the core fall into the sleep state. After that, the micro-benchmark attempts to access all the cache-lines again. We calculate the difference in time it takes to access all the cache-lines again between when the core stays at CC0 (i.e., not flushing caches) and when the core falls into CC6 (i.e., flushing caches) before the accesses. We run the experiment on two processors with different private cache sizes (i.e., E5-2620v4 with 256KB L2 cache and Gold 6134 with 1MB L2 cache). We disable the HW prefetcher; note that the prefetcher can prefetch the flushed cache-lines without the explicit accesses, reducing the cost of cache flushing. As a result, the cost of cache flushing in E5-2620v4 and Gold 6134 processors are 7μs and 26.4μs, respectively. These results are the worst case scenario since a real application typically accesses a part of flushed cache-lines instead of all the flushed cache-lines.

Putting everything together, the maximum wake-up penalty from the deepest sleep state (activation + cache flush penalty) is about 53.8μs (27.4μs + 26.4μs) in Xeon Gold 6134 processor. Since the wake-up penalty from the deepest sleep state is in μs scale, the sleep states do not have a notable impact on the tail latency of memcached which is in milliseconds scale. In Section 6, we will also quantitatively investigate the impact of the sleep state policies along with power management policies for V/F states.

6 EVALUATION

6.1 Methodology

We evaluate NMAP with Intel Xeon Gold 6134 processor consisting of eight cores supporting per-core DVFS with 16 P states ranging from 1.2 GHz (P15) to 3.2 GHz (P0) after disabling hyper-threading technology. We run eight threads for a memcached and nginx server on the eight-core processor. For NMAP, we use 10ms as the timer interval and two thresholds (i.e., *NI_TH* for Network Intensive Mode and *CU_TH* for CPU Utilization based Mode) through our profiling technique described in Section 4.2.

We use an Intel 82599 NIC with ixgbe driver supporting Receive Side Scaling (RSS) technology [18] that distributes network packets across cores. RSS evenly distributes packets in our experimental setup, thus each core handles almost the same amount of network loads. We generate three different load-levels (i.e., low, medium, and high) for memcached and nginx. For memcached, we receive 30K, 290K, and 750K requests per second (RPS) from 20 client threads, respectively. We handle 18K, 48K, and 56K RPS from 20 client threads, respectively, for nginx. We use the D-Link DXS-1210-12SC 10GbE switch to connect the server and client. We utilize the Running Average Power Limit (RAPL) counter that maintains the energy consumed by a processor package.

For comparison, we use three governors ondemand and performance offered by cpufreq driver of Linux, and intel_powersave governor that is default governor of intel_pstate driver for Intel processors. The ondemand and the intel_powersave governors

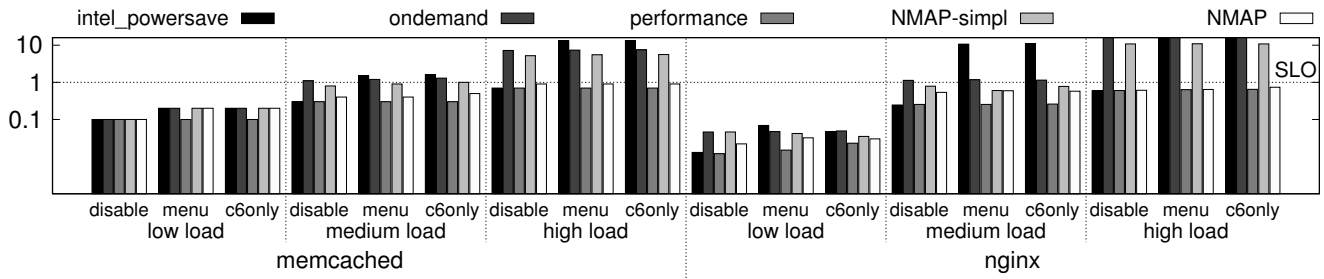


Figure 12: Comparison of P99 latency.

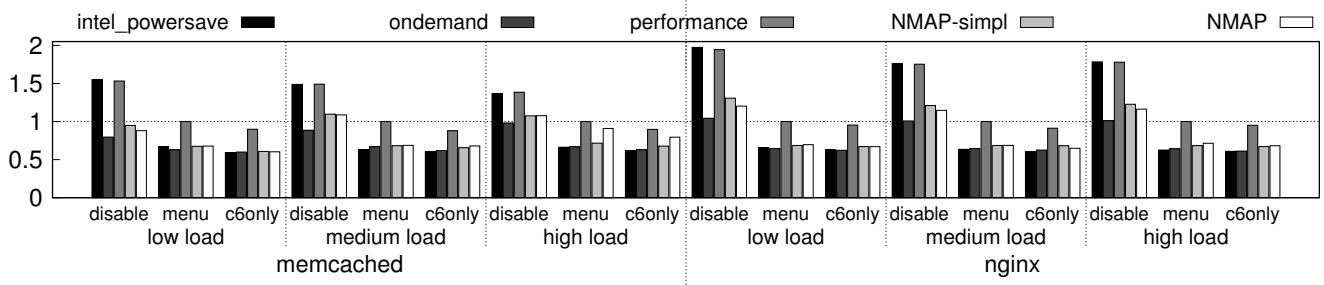


Figure 13: Comparison of energy consumption.

dynamically adjust the current V/F based on the CPU utilization while the `performance` governor statically sets the highest V/F; we use 10ms sampling interval for ondemand and `intel_powersave` governors. We also include the results of three sleep state policies, `disable`, menu, and `c6only` described in Section 5.2.

6.2 Comparison with conventional power management

Fig. 9 plots wake-up time of `ksoftirqd`, P state changes, the number of packets processed in interrupt mode, and the number of packets processed in polling mode with `NMAP`. Unlike the ondemand governor plotted in Fig. 2, `NMAP` maximizes the current V/F at the early part of bursts. Furthermore, `NMAP` lowers the current V/F quickly when the ratio of polling to interrupt decreases, reducing energy consumption. Fig. 10 plots the response latency of every packets during 0.5 second with `NMAP` while Fig. 11 plots the CDF of response latency with the same environment. As shown in Fig. 11, only 0.92% and 0.06% packets with `NMAP` show the longer latency than 1ms and 10ms, which are the SLOs of P99 latency for `memcached` and `nginx`, respectively.

Fig. 12 and Fig. 13 plot P99 latency and energy consumption of `intel_powersave`, ondemand, `performance`, and simplified version of `NMAP` exploiting `ksoftirqd` (denoted as `NMAP-simpl`), and `NMAP`. Those power management policies also co-operate with three different sleep state policies, menu, `disable`, and `c6only`. While `c6only` shows the most energy reduction, the sleep state policy does not make a notable difference in P99 latency. This is because the wake-up penalty of sleep states (i.e., $27{+}\alpha(<26.4)\mu$s) do not have a notable impact on the latency in ms scale as discussed in Section 5.2. Nevertheless, the sleep state management is a challenge for latency-critical applications with μs scale SLOs and we leave that discussion to future work.

As plotted in Fig. 12, the `performance` governor always shows the shortest tail latency among V/F state management policies while showing the most energy consumption since it always sets the highest V/F state. On the contrary, CPU utilization based DVFS governors, `intel_powersave` and ondemand governors, show lower energy consumption than `performance` governor, but they violate the SLOs at medium and high load-levels except for `intel_powersave` with `disable`; note that `intel_powersave` always operates cores at P0 with `disable` since it calculates the CPU utilization based on the residency time at CC0 for the next V/F state. For example, `intel_powersave` governor shows much longer P99 latency than the SLOs with `memcached` and `nginx` by up to 13.1x and 40.6x, respectively, and ondemand governor also shows much longer P99 latency than the SLOs by up to 7.4x and 20.9x. `NMAP-simpl` satisfies the SLOs at medium load. Compared with ondemand governor, `NMAP-simpl` improves P99 latency of `memcached` and `nginx` by up to 33.3% and 31.4% respectively, while consuming more energy by up to 1.8% and 6.5%. For `memcached`, compared with `performance` governor, `NMAP-simpl` reduces energy consumption by 34.8% and 31.8%, respectively, at low and medium load. For `nginx`, compared with `performance` governor, `NMAP-simpl` reduces energy consumption by 31.4% and 31.5%, respectively, at low and medium load. However, since transitioning to the `Network Intensive Mode` in the event of `ksoftriqd` wake-up is not fast enough for high load, it fails to satisfy the SLO at the high load. Meanwhile, `NMAP` satisfies the SLOs at all loads for `memcached` and `nginx` while reducing the energy consumption considerably by quickly reacting to the packet bursts based on the polling to interrupt ratio. For `memcached`, `NMAP` reduces the energy consumption by 35.7%, 31.4%, and 9.1% at low, medium, and high load, respectively, compared with `performance` governor. For `nginx`, `NMAP` reduces the energy consumption by 30.4%, 31.3%, and 28.6%, at low, medium, and high load, respectively, compared with `performance` governor.

6.3 Comparison with state-of-the-arts, SLO-aware power managements

`Rubik` [21] and μ`DPM` [8] perform long-term online profiling through statistical models of latency-critical applications. Based on the profiling results, they adjust the V/F state according to the length of request queues. `Adrenaline` [16] identifies latency-critical requests using application-level hints and sets the appropriate V/F state according to the latency-critical request rate. These studies assume

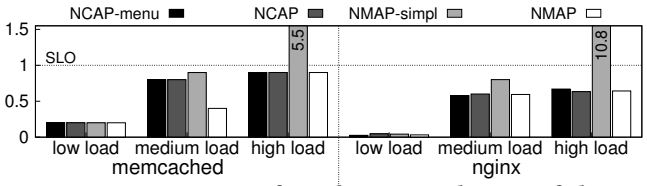


Figure 14: Comparison of P99 latency with state-of-the-art power management.

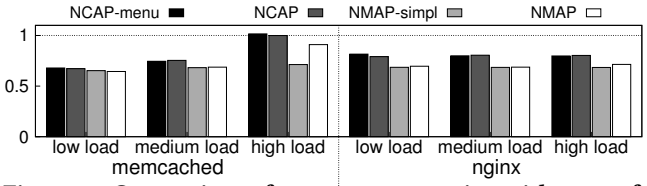


Figure 15: Comparison of energy consumption with state-of-the-art power management.

that *fast V/F state transition* is supported to set the appropriate V/F state in response to network loads. However, as discussed in Section 5.1, the commercial processors do not support such fast V/F transition within several or tens of μs.

NCAP identifies latency-critical requests at programmable NICs. It periodically measures network loads and maximizes the V/F state of all cores when it observes the excessive load. NCAP maximizes the V/F state in the event of network burst and gradually decreases the V/F based on the periodic monitor, it does not require the very short latency for V/F state transition. Since NCAP is the simulation-based study while requiring HW modification, for comparison with NCAP, we implement a software version of NCAP and tune parameters to satisfy the SLOs at a high load of each application; the software version of NCAP has a slightly longer monitoring period than hardware implementation of NCAP. For comparison, we also plot the results of NCAP after enabling menu governor, denoted as NCAP-menu, since the original NCAP disables the sleep state governor when it detects the excessive load. We also use menu governor for NMAP.

Fig. 14 and Fig. 15 show P99 latency and energy consumption of NCAP-menu, NCAP, NMAP-simpl, and NMAP. The results of P99 latency are normalized to the SLOs, and the results of energy consumption are normalized to the energy consumed by performance governor with menu. As plotted, NCAP-menu and NCAP do not show a notable difference in P99 latency and energy consumption. This is because, as discussed in Section 5.2, the processor rarely enters the sleep state in the middle of a request burst.

In terms of P99 latency, NMAP-simpl fails to satisfy the SLO at high load for both memcached and nginx, whereas NMAP and NCAP satisfies the SLO for all loads. NMAP shows further energy reduction than NCAP. NMAP reduces energy by 4.2%, 8.8%, and 9% respectively at low, medium, and high load of memcached compared with NCAP. In case of nginx, NMAP reduces energy consumption by 12%, 14.7%, and 11% compared with NCAP. Although NMAP-simpl violates the SLOs at high load, it also shows further energy reduction compared with NCAP. NMAP-simpl reduces energy consumption by 2.9%, 9.4%, and 28.7% at loads of memcached compared with NCAP. It also reduces energy consumption of nginx by 13.2%, 14.8%, and 14.7% compared with NCAP. These performance benefits are that NMAP operates as a per-core DVFS, thereby providing an opportunity for further energy efficiency improvement, while NCAP operates as chip-wide

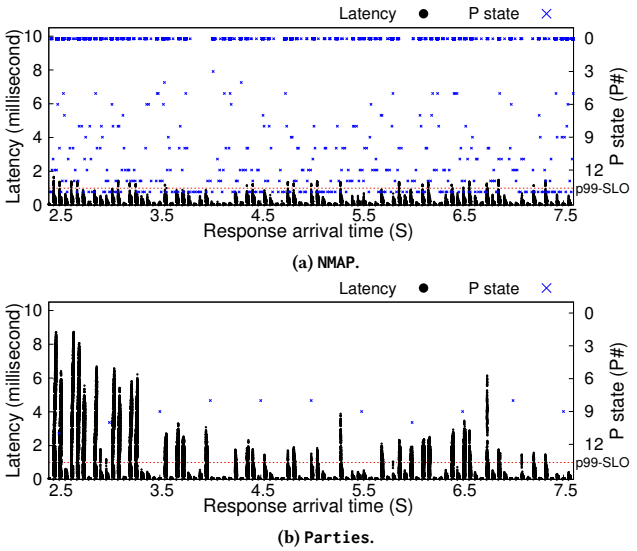


Figure 16: Response latency of every packets and P state changes.

DVFS; NCAP operates based on the total network loads at the NIC while not considering each core's load. Besides, NMAP, unlike NCAP, does not require additional hardware, and can be easily integrated and applied to the commodity systems.

Lastly, we evaluate NMAP with a workload where the load is changing while comparing NMAP with a long-term DVFS approach, Parties [7]; note that NMAP does not require to reset the threshold values as the load changes. We choose a load among the low, medium, and high load at random and change the load periodically while running memcached. Parties monitors the tail latency periodically and adjusts the V/F state based on the slack that is the difference between the SLO and the measured latency. Such feedback-based techniques [34, 35] typically have relatively long decision-making interval since they obtain tail response latency from clients; Parties decides the V/F state every 500ms. Consequently, such long-term DVFS techniques will fail to react to sudden request bursts due to their long interval.

Fig. 16 plots the changes in V/F state and response latency of every packets for 5 seconds with NMAP and Parties. NMAP maximizes the V/F state for request bursts while not violating SLO even with the workload where the load is changing; only 0.18% of requests with NMAP show the longer latency than the target SLO. On the other hand, Parties sets the V/F states not enough to handle the load (e.g., P8) while reacting to the load every 500ms. Consequently, 26.62% of requests with Parties show the longer response latency than the target SLO.

7 RELATED WORK

Providing energy proportionality using power management techniques is important for reducing the Total Cost of Ownership (TCO) of data-centers as the data-center servers often show low average resource utilization (10~50% utilization across different resources) [4, 12, 22, 28, 43]. DVFS is a representative technology that provides energy-proportionality to a processor, and many prior DVFS studies [13, 15, 23, 32, 42] propose an approach to improve the energy efficiency (e.g., EDP (Energy Delay Product)) of CPU

intensive applications. Recently, with the prevalence of latency-critical applications, many studies propose approaches to improve the energy efficiency through DVFS while satisfying SLOs of the latency-critical applications. We classify existing DVFS studies for latency-critical applications into short-term DVFS and long-term DVFS according to the period of V/F state adjustment.

Short-term DVFS. Adrenaline [16] and TimeTrader [41] propose DVFS policies that adjust the V/F state for each request. Adrenaline first calculates the interval between latency-critical requests through an application-level hint and adjusts the V/F state according to the interval. TimeTrader measures the latency of each request and adjusts the V/F state according to the slack, which is the difference between the latency and the target SLO. Rubik [21] and μDPM [8] propose and use the statistical model for setting the minimum frequency under the given SLO constraints. They show the improvement of energy efficiency through the evaluation on a simulated per-core DVFS environment. However, in order to apply these approaches to current latency-critical applications with high and varying RPS, there is a limitation that the processor must support a very short *re-transition latency* in μs scale. On the contrary, NMAP proposes the more coarse-grained, but practical mechanism implemented in NAPI. Consequently, NMAP can be readily applied to the commercial server processors (e.g., Intel Xeon processor) requiring hundreds of microsecond scale *re-transition latency*. NCAP [1] identifies latency-critical requests (e.g., GET requests) through a programmable NIC and calculates their RPS. If the calculated RPS value exceeds a certain threshold, NCAP disables the sleep states and maximizes the V/F state of all cores. Otherwise, the CPU utilization based DVFS governor is used. It improves energy efficiency under the SLO constraints for the chip-wide DVFS environments. To monitor the network bursts, NCAP requires the hardware modification and shows the efficiency through full-system simulation. NMAP considers not only packet loads but also the current packet processing status of each core operating at a specific P state for power management without hardware modification. Furthermore, NMAP inherently supports per-core DVFS processors since each core processes packets through NAPI individually and simultaneously.

Long-term DVFS. PEGASUS [25] periodically measures the latency of an application and the computing power of the node – instead of CPU utilization – and adjusts V/F state by applying a long-term feedback controller based on the measured latency and power. SleepScale [24] determines the optimal V/F and sleep states by predicting the number of requests for future latency-critical applications every 1 second. PowerChief [44] checks the bottleneck of the application through the average queuing time and service time of requests for latency-critical applications every few seconds, and determines the number of instances of the application or the increase of the V/F state. Twig [35] and Hipster [34] periodically monitor application performance (e.g., latency) and Hardware Performance Counter to improve the energy efficiency of latency-critical applications, and based on the monitored values, use reinforcement learning to adjust core allocation and DVFS. Twig targets homogeneous architecture while Hipster targets heterogeneous architecture (e.g., big.LITTLE architecture). Co-PI [20] periodically co-adjusts the interrupt generation frequency and V/F state according to the table generated by offline profiling for energy efficiency of latency-critical applications. Heracles [26] and Parties [7] address resource allocation problems when deploying latency-critical applications and best effort applications with low priority on a single node. They control not only DVFS but also the allocation of various computing resources such as core allocation and cache partitioning in the long term. Additionally, Parties considers scenarios in which two or more latency-critical applications are deployed together on a node. IX [5] proposes a core allocation policy along with power management running on the top of the customized OS that optimizes network performance. For latency-critical workloads, IX adjusts the number of cores first, and then manipulates the V/F state of all cores, i.e., chip-wide DVFS, every 100ms based on the network queue for an application. Since the aforementioned studies do not quickly react to load changes on a server, it is likely to become inefficient for servers running latency-critical applications with rapidly changing loads.

Power management exploiting sleep state. There have been studies [2, 9, 31, 40, 45] that reduce energy consumption while guaranteeing SLO of latency-critical applications using only the sleep states. They deal with core scheduling for the application [2, 45], modify the sleep state governor [40], or delay the processing of requests [9, 31], so that each core of the processor can stay in the deep sleep state for a long time. In our experimental environment with a few ms scale SLO, the sleep state does not show a huge difference in tail latency, but we expect a more sophisticated sleep state management in an environment that requires a few tens of microsecond scale SLO [3]. We leave it as future work to consider the sophisticated use of sleep state integrated with DVFS.

8 CONCLUSION

In this paper, we propose NMAP, **N**etwork packet processing **M**ode-**A**ware **P**ower management that leverages NAPI modes in the Linux network software stack to perform SLO-aware, per-core power management for latency-critical servers. We first investigate the behavior of NAPI and demonstrate the limitations of the current CPU utilization based power management techniques. Second, we analyze the characteristics of power management on commercial processors and show the limitations of previously proposed short-term DVFS techniques. Lastly, we propose and implement NMAP that piggybacks on the existing NAPI mechanism in Linux to quickly identify load surges on each core and adjust the V/F state of the core accordingly. NMAP comes in two flavors: a simplified version that triggers V/F increase based on the sleep/wake-up events of ksoftirqd; and a more sophisticated version that makes decisions based on the ratio of polling to interrupt in NAPI. Our experimental results show that NMAP satisfies SLOs while consuming significantly lesser energy compared to performance governor and state-of-the-art power management, NCAP.

ACKNOWLEDGMENTS

This work was partly supported by National Research Foundation of Korea (NRF) grants funded by the Korean government (MSIT) (NRF-2020R1C1C1013315, NRF-2018R1A5A1060031), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2014-3-00065, Resilient Cyber-Physical Systems Research), and National Science Foundation grant (CNS-1705047). Daehoon Kim is the corresponding author.

REFERENCES

[1] Mohammad Alian, Ahmed HMO Abulila, Lokesh Jindal, Daehoon Kim, and Nam Sung Kim. 2017. NCAP: Network-Driven, Packet Context-Aware Power Management for Client-Server Architecture. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 25–36.

[2] Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, and Timothy Zhu. 2020. Peafowl: in-application CPU scheduling to reduce power consumption of in-memory key-value stores. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*. 150–164.

[3] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

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

[5] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems (TOCS)* 34, 4 (2016), 1–39.

[6] Dominik Brodowski and Nico Golde. 2002. CPU frequency and voltage scaling code in the Linux kernel. *https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt* (2002).

[7] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 107–120.

[8] Chih-Hsun Chou, Laxmi N Bhuyan, and Daniel Wong. 2019. μDPM: Dynamic power management for the microsecond era. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 120–132.

[9] Chih-Hsun Chou, Daniel Wong, and Laxmi N Bhuyan. 2016. Dynsleep: Fine-grained power management for a latency-critical data center application. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. 212–217.

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

[11] Pierre Delforge. 2015. America's Data Centers Consuming and Wasting Growing Amounts of Energy. [Online]. Available: https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy.

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 127–144.

[13] Stijn Eyerman and Lieven Eeckhout. 2010. A counter architecture for online DVFS profitability estimation. *IEEE Trans. Comput.* 59, 11 (2010), 1576–1583.

[14] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 124 (2004).

[15] Rong Ge, Xizhou Feng, and Kirk W Cameron. 2005. Improvement of power-performance efficiency for high-end computing. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 8–pp.

[16] Chang-Hong Hsu, Yunqi Zhang, Michael A Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, and Ronald G Dreslinski. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 271–282.

[17] Intel Intel. 2014. 82599 10 GbE Controller Datasheet.

[18] Intel Intel. 2016. Receive-Side Scaling (RSS). *http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf* (2016).

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

[20] Ki-Dong Kang, Hyungwon Park, Gyeongseo Park, and Daehoon Kim. 2020. Co-Adjusting Voltage/Frequency State and Interrupt Rate for Improving Energy-Efficiency of Latency-Critical Applications. *IEEE Access* 8 (2020), 201028–201039.

[21] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast analytical power management for latency-critical systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 598–610.

[22] Harshad Kasture and Daniel Sanchez. 2014. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 729–742.

[23] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. 2010. Interval-based models for run-time DVFS orchestration in superscalar processors. In *Proceedings of the 7th ACM international conference on Computing frontiers*. 287–296.

[24] Yanpei Liu, Stark C Draper, and Nam Sung Kim. 2014. Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 313–324.

[25] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 301–312.

[26] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 450–462.

[27] Peter Macken, Marc Degrauwe, Mark Van Paemel, and Henri Oguey. 1990. A voltage reduction technique for digital systems. In *In IEEE International Conference on Solid-State Circuits*. 238–239.

[28] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 248–259.

[29] David Meisner, Brian T Gold, and Thomas F Wenisch. 2009. PowerNap: eliminating server idle power. In *Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 205–216.

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

[31] David Meisner and Thomas F Wenisch. 2012. DreamWeaver: architectural support for deep sleep. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 313–324.

[32] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N Patt. 2012. Predicting performance impact of DVFS for realistic memory systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 155–165.

[33] Jeffrey C Mogul and KK Ramakrishnan. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS)* 15, 3 (1997), 217–252.

[34] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. Hipster: Hybrid task manager for latency-critical cloud workloads. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 409–420.

[35] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. 2020. Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 167–179.

[36] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. 2007. cpuidle: Do nothing, efficiently. In *Proceedings of the Linux Symposium*, Vol. 2. Citeseer, 119–125.

[37] Venkatesh Pallipadi and Alexey Starikovskiy. 2006. The ondemand governor. In *Proceedings of the Linux Symposium*.

[38] Will Reese. 2008. Nginx: the high-performance web server and reverse proxy. *Linux Journal* (2008), 2.

[39] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. 2001. Beyond Softnet. In *Annual Linux Showcase & Conference*, Vol. 5. 18–18.

[40] Erfan Sharafzadeh, Seyed Alireza Sanaee Kohroudi, Esmail Asyabi, and Mohsen Sharifi. 2019. Yawn: A CPU Idle-state Governor for Datacenter Applications. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*. 91–98.

[41] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and TN Vijaykumar. 2015. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 585–597.

[42] Qiang Wu, Vijay J Reddi, Youfeng Wu, Jin Lee, Dan Connors, David Brooks, Margaret Martonosi, and Douglas W Clark. 2005. A dynamic compilation framework for controlling microprocessor energy and performance. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 271–282.

[43] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 607–618.

[44] Hailong Yang, Quan Chen, Moeiz Riaz, Zhongzhi Luan, Lingjia Tang, and Jason Mars. 2017. Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 133–146.

[45] Xin Zhan, Reza Azimi, Svilen Kanev, David Brooks, and Sherief Reda. 2016. Carb: A c-state power management arbiter for latency-critical workloads. *IEEE Computer Architecture Letters* 16, 1 (2016), 6–9.