NCAP: Network-Driven, Packet Context-Aware Power Management for Client-Server Architecture

Mohammad Alian[1], Ahmed H.M.O. Abulila[1], Lokesh Jindal[2], Daehoon Kim[1,3], Nam Sung Kim[1]
[1]University of Illinois, Urbana-Champaign; [2]University of Wisconsin, Madison; [3]DGIST
{malian2, abulila2}@illinois.edu; lokeshjindal15@cs.wisc.edu; dkim@dgist.ac.kr

Abstract
The rate of network packets encapsulating requests from clients can significantly affect the utilization, and thus performance and sleep states of processors in servers deploying a power management policy. To improve energy efficiency, servers may adopt an aggressive power management policy that frequently transitions a processor to a low-performance or sleep state at a low utilization. However, such servers may not respond to a sudden increase in the rate of requests from clients early enough due to a considerable performance penalty of transitioning a processor from a sleep or low-performance state to a high-performance state. This in turn entails violations of a service level agreement (SLA), discourages server operators from deploying an aggressive power management policy, and thus wastes energy during low-utilization periods. For both fast response time and high energy-efficiency, we propose NCAP, **N**etwork-driven, packet **C**ontext-**A**ware **P**ower management for client-server architecture. NCAP enhances a network interface card (NIC) and its driver such that it can examine received and transmitted network packets, determine the rate of network packets containing latency-critical requests, and proactively transition a processor to an appropriate performance or sleep state. To demonstrate the efficacy, we evaluate on-line data-intensive (OLDI) applications and show that a server deploying NCAP consumes 37~61% lower processor energy than a baseline server while satisfying a given SLA at various load levels.

1. Introduction
In client-server architecture, when servers receive requests sent from clients, the servers process the requests and send back responses to the clients. In particular, for OLDI applications such as web search, it is essential for servers to reduce high-percentile response time and satisfy a given SLA [1]. At the same time, it is critical for these servers to improve their energy efficiency [2].

The processor is the most power-consuming component even in servers with many DRAM modules (DIMMs). For example, the processors in a two-socket server with 16 DIMMs consumes 42% of total server power [3]. To maximize energy efficiency, therefore, it is important for a power management policy [4, 5] to fully exploit various performance and sleep states supported by modern processors [6]. Depending on the current performance demand, cores in a processor can operate at various performance states by increasing or decreasing their voltage/frequency (V/F). Moreover, idle cores in a processor can transition to various sleep states by turning off their clock, decreasing their V to a level that barely maintains their architectural states after turning off their clock, or turning off both their clock and power supply.

Transitioning a processor core from a sleep or low-performance state to a high-performance state, however, incurs a significant performance penalty. If we account for the overhead of system software layers associated with these transitions, the performance penalty is even higher [7]. Such a notable performance penalty can substantially increase high-percentile response time and discourages server operators from deploying an aggressive power management policy that frequently transitions processor cores to a low-performance or sleep state [8, 9, 10, 11].

It is intuitive that the rate of network packets from clients can significantly affect the utilization, and thus performance and sleep states of processor cores in servers. For example, as a server suddenly receives many network packets containing latency-critical requests from clients, its processor cores need to operate at a high-performance state so that it can process the requests and send responses back in time. However, if necessary processor cores have been in a sleep or low-performance state, the server needs to transition these processor cores to a high-performance state. If a server occasionally receives only a few network packets enclosing latency-critical requests from clients, it should transition unnecessary processor cores to a low-performance or sleep state.

In this paper, we propose NCAP, **N**etwork-driven, packet **C**ontext-**A**ware **P**ower management for client-server architecture. Specifically, we first show a strong correlation between the rate of received/transmitted network packets and the performance/sleep state of processors in servers after analyzing the complex interplay between them.

Second, we propose to enhance a NIC and its driver such that NCAP can (1) examine received/transmitted network packets; (2) detect latency-critical requests in the network packets; (3) speculate the completion of requested services; (4) predict an appropriate processor performance or sleep state; and (5) proactively transition a processor to an appropriate performance or sleep state. Especially, NCAP overlaps a large fraction of a notable performance penalty of transitioning processor cores to a high-performance state with a long latency of transferring received network packets from a NIC to the main memory. Consequently, NCAP allows server operators to deploy an aggressive power management policy without notably increasing the high-percentile response time. Note that NCAP does not simply respond to a high rate of any

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.57

network packets (*e.g.*, network packets associated with virtual machine migrations and storage server operations), as it selectively considers latency-critical network packets.

Lastly, we demonstrate the effectiveness of NCAP for two representative OLDI applications with notably different characteristics: `Apache` and `Memcached` at various load levels using an enhanced full-system simulator. To establish the SLA, we take a baseline server that always operates its processor cores at the highest performance state, and measure its 95th-percentile response time at a high-load level [12]. At medium- to high-load levels, a server deploying NCAP consumes 37~61% lower processor energy than the baseline server, while satisfying the SLA. At low- to medium-load levels, it consumes 21~49% lower processor energy than a server employing the most energy-efficient, SLA-satisfying power management policy amongst the current power management policies supported by Linux.

The remainder of this paper is organized as follows. Section 2 describes the background on processor power management and network stack. Section 3 demonstrates a strong correlation between network packet rate and processor power management. Section 4 presents NCAP in detail. Section 5 describes our evaluation methodology. Section 6 evaluates the efficacy of NCAP. Section 7 provides some discussions. Section 8 describes the related work. Section 9 concludes.

2. Background

2.1 Processor Power Management

For power management, processors support performance (P) and sleep (C) states that are interfaced with the OS by Advanced Configuration and Power Interface (ACPI) [6].

P state. The deeper the P state is, the lower the power consumption is at the expense of lower performance. A core in P0 state operates at the highest V/F point that offers the maximum sustainable performance under thermal and power constraints. The current Linux kernel offers three static P-state management policies (*i.e.*, `performance`, `powersave`, and `userspace` governors), and one dynamic P-state management policy (*i.e.*, `ondemand` governor) [4]. Amongst these policies, the `performance` governor always operates cores at P0 state, whereas the `powersave` governor always operates cores at the deepest P state (*i.e.*, lowest V/F point). Lastly, the `userspace` governor enables a user to set the P state of processor cores. In contrast, the `ondemand` governor periodically adjusts the P state based on the utilization of cores.

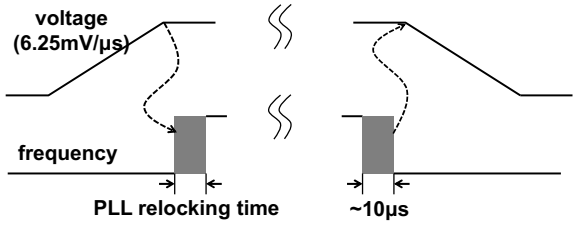

Figure 1: An example of V/F changes and associated performance penalty due to PLL relocking time. The shaded region depicts the duration that a processor core must halt.

Figure 1 illustrates a typical sequence of changing the P state (V/F) of a core. To increase V/F, V is ramped up to a target level at the rate of 6.25mV/µs before the frequency is raised [13]. To decrease V/F, F is reduced before V is decreased. In Intel *i*7-3770 processors, for example, a transition from the lowest to highest V/F takes much longer time (~50µs) than a transition from the highest to lowest V/F (~5µs) [7], because of the latency of ramping up V before raising F. In both cases, the core must halt for ~5µs (*i.e.*, PLL re-locking time) while changing F. Note that F becomes unpredictable and unstable when the PLL attempts to relock the feedback loop of its oscillator to another F level [14].

Figure 2 shows the 95th-percentile latency of Apache at three load levels and for various periods of invoking the ondemand governor. See Section 5 for our detailed evaluation methodology. As the minimum invocation period for the ondemand governor is hard-coded to 10ms in the Linux kernel, we recompiled the Linux kernel after changing the minimum period to 1ms. As shown, the best invocation period varies under different load levels and reducing the invocation period does not always improve the response time due to the performance penalty of frequently invoking the ondemand governor and changing V/F. This is the key reason that the minimum invocation period is hard-coded to 10ms [15].

C state. C0, C1, C3, and C6 states denote idle, halt, sleep, and off states. The deeper the C state is, the lower the power consumption is at the expense of higher performance penalty due to longer wake-up latency. The current Linux kernel provides two C-state management policies (*i.e.*, `ladder` and `menu` governors [5]). The `ladder` governor first transitions a processor core to C1 state and then a deeper C state if the sleep time was long enough. The `menu` governor records how long a processor core has been in a C state in the past and predicts how long it will stay in the C state in the future. Then, it chooses the most energy-efficient C state based on the prediction and wake-up penalty. Currently, the `menu` governor is used by default.

The current Linux kernel invokes `cpu_idle_loop` when the `run_queue` does not have any schedulable job. This function consists of an infinite while loop that repeatedly checks whether or not the `run_queue` has any schedulable job. If there is any newly arrived job, the scheduler is invoked and the jobs in the `run_queue` are executed after being prioritized by the scheduler's policy. Otherwise, the control is

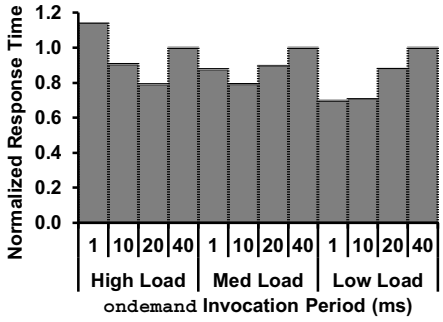

Figure 2: 95th percentile latency of `Apache` for various invocation periods of the `ondemand` governor.

delegated to the C-state governor (*e.g.*, `menu` governor) to reduce the power consumption of idle cores. In C0 state the core waits for a job dispatched to the `run_queue` while executing `NOP` in a kernel while loop. The C-state governor is to apply a chosen C-state to a core based on its policy. To transition the core from the C state, `MWAIT` and `MONITOR` are used in x86 architecture, as cores in C1 – C6 states cannot check whether or not there is a job to do. `MONITOR` arms the address monitoring hardware using an address region specified in `EAX`. When a store occurs in the specified region, it transitions the core to a P state. One of C states is denoted by an specific number as a parameter to `MWAIT`. As `MWAIT` and `MONITOR` are privileged (level-0) instructions, they can be executed only in the kernel space and incur a performance penalty of 6–60µs in Intel *i*7-3770 processors [16].

2.2 Network Stack

TCP/IP is the most widely used communications protocol for high-performance computing despite its well-known overheads. The Ethernet, as the backbone of a datacenter network, is tightly coupled with the TCP/IP layers. The 10Gb Ethernet has been already deployed in modern datacenters and is shown to be effective to bridge the bandwidth gap between the 1Gb Ethernet and other more expensive counterparts such as InfiniBand [17] and Myrinet [18]. However, in addition to high bandwidth, low latency is desired to satisfy a given SLA for OLDI applications. The major contributor to the end-to-end TCP/IP packet latency is the network software layers and multiple long-latency PCIe transactions to deliver a received packet from a NIC to the main memory and processor, as illustrate in Figure 3.

More specifically, the sequence of receiving a packet from a NIC is as follows [19]. (1) Before receiving any packet, the NIC driver creates a descriptor (or ring buffer) in the main memory (`rx_desc_ring` in Figure 3), which contains the metadata of received packets, and initializes the descriptors to point to a receive kernel buffer (`skb` in Figure 3). Subsequently, the NIC driver informs the NIC DMA engine of the start address of `rx_desc_ring`. (2) When a packet is

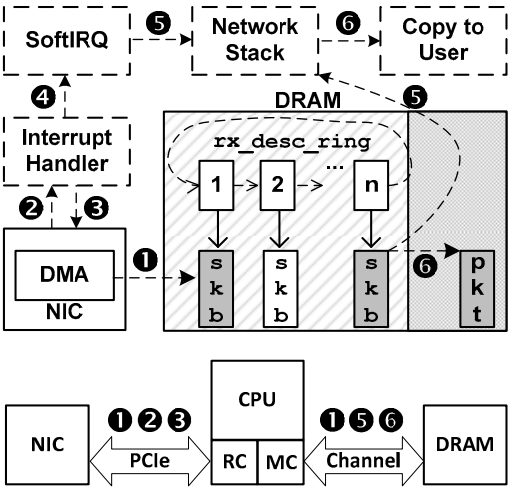

Figure 3: The sequence of receiving a packet from a NIC.

received, based on the descriptor information, the NIC performs a DMA transfer to copy the packet to the associated `skb` (❶ in Figure 3). (3) The NIC will generate a hardware interrupt to the processor. The interrupt handler of the NIC examines the NIC to determine the interrupt cause, which is done by reading a NIC register called Interrupt Cause Read (ICR) register through a PCIe bus (❷ and ❸ in Figure 3). Some NICs use interrupt moderation technique to reduce the number of interrupts posted to a processor by coalescing several hardware interrupts to one interrupt. Although this technique reduces the load on the processor, it increases the end-to-end latency of delivering each packet [20]. (4) After identifying the cause of the interrupt, the interrupt handler enqueues the request for processing the received packet and schedules a SoftIRQ (❹ in Figure 3). (4) The SoftIRQ handler passes the received packet's `skb` to higher layers in network stack and reallocates another `skb` for the used descriptor (❺ in Figure 3). (6) The packet will be copied into a user space buffer after it is processed by the software layers in the network stack (❻ in Figure 3).

For NCAP, we leverage the latency of step ❶, ❷ and ❸ to hide the performance penalty of transitioning cores from a sleep or low-performance state to a high-performance state. Our experiment running `Apache` shows that these step consume 86µs on average.

3. Correlation between Network Activity and Processor Power Management

In many cases, network packets received by a server contain requests to be processed by processor cores. Thus, as a server receives more network packets, the processor utilization will increase. For example, suppose a client sends a request to an OLDI application server. As Hypertext Transfer Protocol (HTTP) requests are encapsulated in TCP packets, the request should go through the server network layers before a processor core in the server can start to process the request. Subsequently, the application will decode the request, bring the requested values from the main memory and send them to the client through one or more TCP packets. Executing the key OLDI processing code and network software layers (for both receiving requests and transmitting responses) increases the processor utilization, when the server receives a burst of requests. Thus, we hypothesize that the rate of received and transmitted network packets substantially affects the power management of the processor, as the processor utilization typically determines the P and C states of the processor.

To demonstrate the correlation amongst the rate of network packets, processor utilization, and dynamic power management policies, we developed an enhanced `gem5` [21] and run `Apache`. The enhanced `gem5` is a full-system simulator that models functions and cycles of the sub-systems (CPU, NIC, DMA, and HDD) of a computer; simulates many computers (or nodes) connected with a given Ethernet topology; and runs Linux on each node. See Section 5 for our detailed evaluation methodology.

In Figure 4 we simulate fifteen clients and one server running `Apache`, and measure server's (1) network transmit and receive bandwidth utilizations (denoted by BW(Tx) and

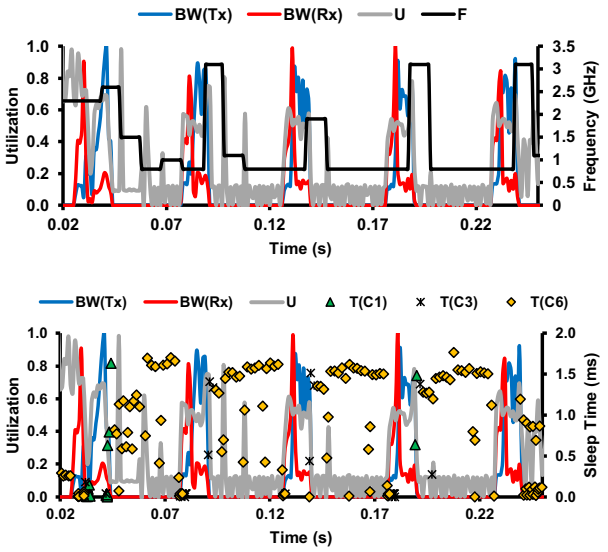

Figure 4: (a) Network bandwidth (BW(Tx) and BW(Rx), core utilization (U), and resulting core frequency (F). (b) core time spent in C1, C2, and C3 states (T(Cx)).

BW(Rx)); (2) processor core utilization (U); (3) processor core frequency (F); and (4) processor core time spent in C1, C3, and C6 states (T(C1), T(C3) and T(C6)). BW(Rx) and BW(Tx) at each point are normalized to the maximum BW(Rx) and BW(Tx) during the entire application run, respectively. The `ondemand` governor dynamically changes P state (F) every 10ms (*i.e.*, the minimum period supported by the `ondemand` governor).

Figure 4(a) shows that a burst of HTTP requests from clients causes a surge in BW(Rx), leading to increase in U and eventually a surge in BW(Tx) for sending the requested responses. The increase in U is due to processing (1) the received and transmitted packets in the network software layers and/or (2) the requests by the clients. One core processes received network packets while another core can process requests. In this experiment, we observe that the `ondemand` governor does not immediately react to a sudden increase in U, because it can increase F only at the end of every 10ms period to amortize the performance penalty of invoking the `ondemand` governor and changing F (*cf.* Section 2.1). Furthermore, the `ondemand` governor increases F only after detecting high U in the previous period. Consequently, if the previous period exhibits low U, there is a significant delay in increasing F (*i.e.*, the maximum delay of up to the period of invoking by the `ondemand` governor), significantly increasing the response time.

Analyzing the burst of BW(Rx) marked with a dotted box in Figure 4(a), we observe the surge of BW(Rx) for 6ms at time 177ms (and that of U shortly after that of BW(Rx)). Subsequently, we observe the surge of BW(Tx) for 9ms at time 181ms. The average BW(Rx), BW(Tx), and U during this period is 42%, 47%, and 48%, respectively. A perfect `ondemand` governor would have boosted F at time 177ms, kept F high for 14ms, and then reduced F at time 191ms. However, the default `ondemand` governor in Linux increases F from 0.8 to 3.1 GHz at 188ms (*i.e.*, 11ms late) and reduces F at 198ms (*i.e.*, 7ms late).

Figure 4(b) shows that entering C states is also highly correlated with the bursts of HTTP requests and their durations. During the idle period between two request bursts, the processor core often transitions to C6 state to reduce power consumption. This shows that the `menu` governor is effective in transitioning a processor core to a C state when the processor core has been idle for a certain period. However, as depicted in Figure 4(b), the processor core frequently transitions to C6 state before a surge of BW(Rx). That is, the processor core for processing network packets (and possibly requests) is in a C state. Thus, the `menu` governor needs to transition the processor core from the C state to a P state before executing any code. Furthermore, some transitions to C states are very short during the surges of BW(Rx). These short transitions to C states can hurt the energy efficiency of the processor [11]. Analyzing the BW(Rx) surge marked with a dotted box in Figure 4(b), we see that core 0, 1, 2, and 3 are in C3, C6, C6, and C6 for 2µs, 337µs, 111µs, and 2.12ms, respectively. At the very beginning of the surge, the cores transition to C3 and C6 states 10 and 5 times, and stay in these C states for 30µs and 31µs, respectively, on average. After 2ms from the beginning of the surge period, the `menu` governor does not transition the processor cores to C states anymore until the BW(Rx) and subsequent BW(Tx) surging periods end.

The rate of network packets is inherently unpredictable at the low- to medium-levels of request rates (or simply load levels). That is, the network packet rate can suddenly increase and decrease after it stays at a low level for a long period [20]. On the other hand, as discussed in Section 2.1, it takes long time to a server to transition processor cores from a deep C or P state to the P0 state before it can process received requests [11]. This increases high-percentile response time for next bursts of requests, and thus may entail SLA violations. Consequently, server operators may simply deploy the `performance` governor that always operates processor cores at P0 state, wasting energy at low- to medium-load levels.

4. NCAP Management

In this section, we propose NCAP, **N**etwork-driven, packet **C**ontext-**A**ware **P**ower management for client-server architecture based on our observations from Section 3. More specifically, NCAP, which aims to assist the `ondemand` and `menu` governors, leverages a low-level network packet context to proactively transition cores to an appropriate P or C state. This can significantly improve both response time and energy efficiency compared with the default `ondemand` and `menu` governors.

Figure 5 depicts the key aspects of NCAP comprised of an enhanced NIC and its driver. In the enhanced NIC (Figure 5(a)), `ReqMonitor` (Figure 5(b)) and `TxBytesCounter` observe received and transmitted network packets. If `ReqMonitor` and `TxBytesCounter` detect a significant increase in the rate of received network packets (encapsulating latency-critical requests) and decrease in the rate of transmitted network packets, `DecisionEngine` (Figure 5(c))

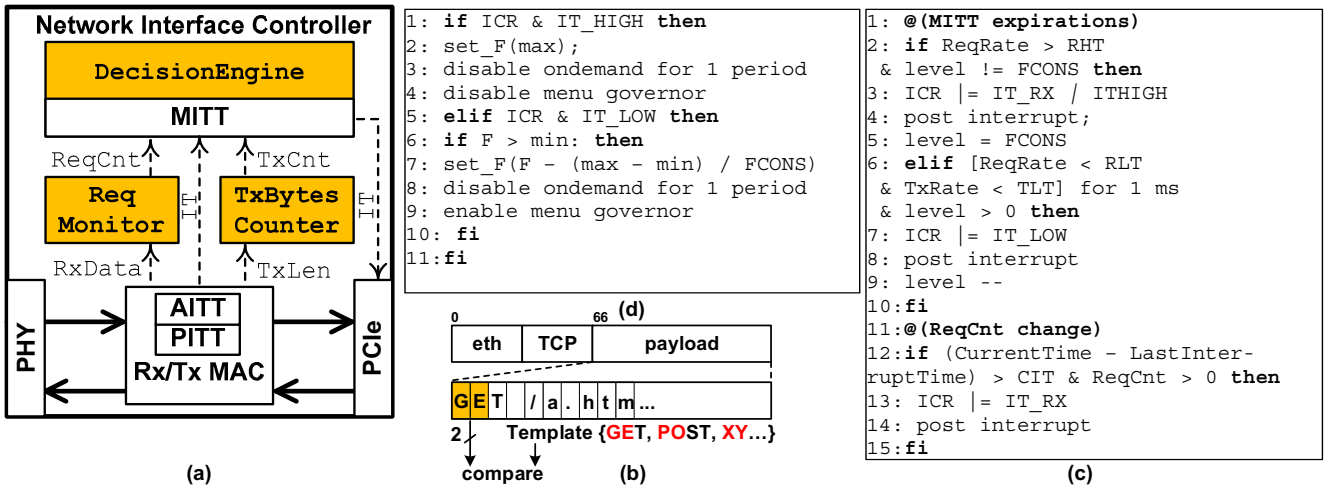

Figure 5: NCAP comprised *of* (a) enhanced NIC; (b) `ReqMonitor` in NIC; (c) `DecisionEngine` in NIC; and (d) an enhanced NIC interrupt handler driver.

triggers a special interrupt sent to the processor. Subsequently, the enhanced interrupt handler of the NIC driver running on a processor core (Figure 5(d)) proactively transitions necessary processor cores to the P0 state (*i.e.*, the highest-performance state) if the processor cores have been in low-performance or sleep states. Furthermore, if a request is received and `DecisionEngine` observes a long interval between the past interrupts and current times, it speculates that the processor cores are in C states and immediately generates an interrupt to proactively transition these processor cores to higher-performance state. The immediate C and P state changes for such events allow a server deploying NCAP to quickly service a large number of requests abruptly sent from clients while consuming lower energy than a server adopting the default `ondemand` and `menu` governors.

4.1 Context-Aware Packet Rate Detection

As observed in Section 3, it is likely that OLDI applications need high performance right after a surge in the rate of network packets enclosing latency-critical requests received by servers. A naïve approach to respond to such an event is to transition processor cores from their C states to the P0 state as soon as the rate of any received network packets exceeds a certain threshold value. Such a naïve approach, however, has some limitations.

First, certain types of network packets received by servers are not latency-critical. One example is a packet containing a request to update the content of a web page (*e.g.*, a `PUT` request in HTTP). Another example is network packets of off-line data analytics applications that consume high network bandwidth but do not have SLAs to satisfy. Second, the length of network packets containing latency-critical requests is often very short [22], and thus the aggregate size of a burst of such network packets may not surpass the threshold value set to operate processor cores at P0 state. On the contrary, the naïve approach may unnecessarily transition processor cores to the P0 state, as simply observing the received network packet rate lacks a context (*i.e.*, whether or not the received network packets are latency-critical).

The requests that are generated by OLDI applications typically have a predefined format, following a standardized universal protocol. For instance, HTTP is a unified application protocol that is widely used for OLDI applications. An HTTP request starts with a request type (*e.g.*, `GET`, `HEAD`, `POST`, or `PUT`) which is followed by a requested URL, and other request header fields [23]. To proactively transition processor cores from a deep C or P state to the P0 state, instead of simply using the received packet rate as a hint, we exploit the fact that latency-critical requests of OLDI applications often have a predefined format.

To detect latency-critical requests, we propose `ReqMonitor` (Figure 5(b)) in an enhanced NIC. Most online requests are encapsulated in a TCP packet. The payload field, which includes a request, starts from the 66th byte of a received TCP packet. `ReqMonitor` compares the first two bytes of the payload with a set of templates that are stored in some registers in a NIC. These registers, which are programmable through the operating system's `sysfs` interface [24], can be programmed to store latency-critical request types such as `GET`, when running the initialization subroutine of the NIC driver. Consequently, `ReqMonitor` can determine whether or not a received network packet is a latency-critical one. If so, `ReqMonitor` increments `ReqCnt`.

Furthermore, we observe that the significant decrease in the rate of transmitted network packets subsequently entails to low U and the `ondemand` and `menu` governors eventually transition the processor cores to deep P or C states in Figure 4, as the processor has completed its services for the requests. Thus, we also propose `TxBytesCounter`, which counts the number of transmitted bytes to determine `TxCnt`. The rationale behind using `TxCnt` without any context is that most responses are larger than the Ethernet maximum transmission unit [25], and thus several TCP packets constituting a single response are transmitted. Detecting such a long chain of latency-critical response packets requires a complex hardware. Moreover, even if the transmitted packets are not latency critical, operating the processor at P0 state can complete the packet transmission faster and thus allow the processor cores

to transition to a C state sooner. ReqCnt and TxCnt will be used for the enhanced Master Interrupt Throttling Timer (MITT) to determine ReqRate and TxRate, respectively, the use of which will be discussed in Section 4.3.

4.2 Generation of NCAP Interrupt

In order to prevent a NIC from posting too many interrupts to the processor whenever the NIC receives a packet, NICs employ a set of Interrupt Throttling Timers (ITTs) to moderate the number of interrupts that a NIC generates. This is depicted in Figure 6. More specifically, all the Gigabit Ethernet (GbE) controllers contain five timers to moderate the interrupt rate: two Absolute Interrupt Throttling Timers (AITTs); two Packet Interrupt Throttling Timers (PITTs); and one MITT. The AITT and PITT are triggered by a network event (*i.e.*, whenever a packet is received or transmitted) to limit the maximum number of interrupts posted upon receiving or transmitting packets. In contrast, the MITT operates independently from any interrupt source or network event, and constrains the total interrupt rate of a NIC. That is, an interrupt is posted to the processor when the MITT expires. Before posting an interrupt to the processor, the NIC sets an ICR (*cf.* Section 2.2) with the type of interrupt that it intends to send to the processor from a set of interrupt types predefined by the device driver (*e.g.*, IT_RX when a received packet is ready to be passed to the network software layers).

For NCAP to trigger a transition of a processor core from a deep C or P state to the P0 state at appropriate moments, we propose two more interrupt types, using the unused bits of ICR: IT_HIGH and IT_LOW, respectively. When to trigger IT_HIGH or IT_LOW will be discussed in Section 4.3.

4.3 Decision of P- and C-State Changes

For NCAP to set an ICR and subsequently post an interrupt to the processor at the right moment, we propose DecisionEngine depicted in Figure 5(c). Two events trigger DecisionEngine: (1) MITT expirations and (2) ReqCnt changes. When the MITT expires (at every 40 to 100μs), a new ReqRate is determined by ReqCnt. If ReqRate is greater than a request rate high threshold (RHT) and F is not already set to the maximum (P0 state), then DecisionEngine posts an interrupt to the processor after setting IT_HIGH and IT_RX bits of ICR. On the contrary, if ReqRate and TxRate are smaller than a request rate low threshold (RLT) and a transmission rate low threshold (TLT) for 1ms, DecisionEngine posts an interrupt to the processor after setting IT_LOW bit of ICR. When an interrupt with IT_HIGH and IT_RX is posted, NCAP performs a sequence of actions as follows: (1) increasing F to the maximum frequency; (2) disabling the menu governor; and (3) disabling the ondemand governor for one invocation period. We disable the menu governor to prevent short transitions to a C state during a surge period of BW(Rx) (Figure 4). We also disable the ondemand governor for one invocation period to prevent any conflict between NCAP and ondemand governor decisions. While NCAP sets F to the maximum upon an assertion of IT_HIGH, it can be more conservative in decreasing F (*i.e.*, reducing F to the minimum over several steps). FCONS is a parameter to determine the number of steps to reach the minimum F. That is, the number of required back-to-back interrupts with IT_LOW to reduce F to the minimum. NCAP enables the menu governor when the first IT_LOW interrupt is posted.

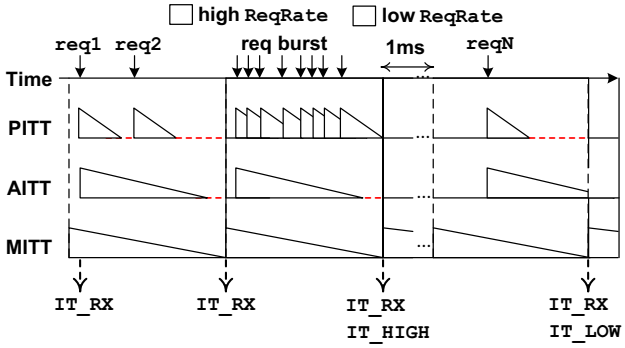

Figure 6: Illustration of NCAP and its interaction with interrupt throttling timers in a NIC.

A change in ReqCnt infers that new requests have been received by the NIC. If the time interval between the current request and the last interrupt posted to the processor (CurrentTime – LastInterruptTime) is larger than the processor idle time threshold (CIT), which is typically set by the user or menu governor, DecisionEngine immediately posts an interrupt with IT_RX to the processor. When the processor has not been interrupted for a long time, NCAP speculates that processor cores have been in an idle state for a while, and thus transitioned to a C state (*cf.* Section 3). In such an event, NCAP immediately sends an interrupt to the processor so that the target processor core to process the request(s) can transition from the C state to active state and gets ready to service the requests.

Figure 5(d) shows the enhancements in the NIC hardware interrupt handler. When an interrupt is received from the NIC, if the IT_HIGH bit of ICR is set, the NIC hardware interrupt handler calls some APIs of the cpufreq driver, which is responsible for changing F in the Linux kernel, to change F to the maximum. Otherwise, if the IT_LOW bit is set, then the NIC hardware interrupt handler determines the next F based on FCONS.

Figure 6 overviews NCAP under a certain packet arrival scenario. Supposed that req1 is received after the NIC has been in a long idle period (longer than CIT). Then, DecisionEngine immediately sends an interrupt with IT_RX to transition a processor core to a P state regardless of the MITT expiration time. Later, when a burst of requests is received and the MITT expires, ReqRate is updated. This triggers DecisionEngine to send an interrupt with IT_HIGH to change F to the maximum and disable menu governor. After detecting a low-activity period of 1ms, DecisionEngine sends one or several interrupts with IT_LOW to decrease F, and enable menu governor again, depending on whether a given policy is aggressive or conservative.

5. Methodology

The current official release of gem5 is a full-system simulator that models functions and cycles of the sub-systems

(CPU, NIC, DMA, and HDD) of a computer, but it supports only thread-based parallelism within a single process. That is, it does not support parallelism across multiple simulation hosts, and thus it can model and simulate only a limited number of nodes. Moreover, it supports only primitive NIC and network switch models, preventing us from precisely capturing various network effects on overall performance of clients and servers connected by a network. Thus, for our study we developed an enhanced `gem5` that supports the simulation of multiple nodes using multiple simulation hosts and synchronization amongst these simulated nodes [21].

To enable the `ondemand` governor in `gem5`, we leverage prior work [6, 26]. More specifically, we enable the P-state controller, a memory-mapped device that provides registers for the ACPI and changes F of simulated cores. The implemented P-state controller with a modified `cpufreq` driver in Linux allows the `ondemand` governor to control the current P state of the simulated cores. For our experiment, we run the `ondemand` governor with an invocation period of 10ms (*i.e.*, the minimum period supported by the default `ondemand` governor [15]).

Furthermore, we implement a `cpuidle` driver for `gem5` to enable the `menu` governor; a `cpuidle` driver conveys the information on available C states to the `menu` governor. Two key parameters, which affect decisions of the `menu` governor, are the exit latency (*i.e.*, latency associated with transitioning from a C state to a P state) and the residency (*i.e.*, the minimum amount of time the processor core should spend in a given C state to make the transition worth the energy penalty). In our experiments, we study the `cpuidle` behavior with three C states, C1, C2, and C3 with exit latency of 2µs, 10µs, and 22µs and residency of 10µs, 40µs, and 150µs, respectively [7]. We model these C states by halting the execution of instructions. Once the `cpuidle` driver commands a core to transition to a C state, the instruction fetch is stalled and it idles as soon as the processor core pipeline is drained. On the other hand, when it is decided to transition an idle processor core to a P state, the processor core resumes to fetch and execute instructions after applying an appropriate delay to model the exit latency.

Table 1: Processor configurations.

Category	Configuration
O3 core	Number of cores: 4 Number of P and C states: 15 and 3 V/F at P states: 0.65V/0.8GHz to 1.2V/3.1GHz Processor max power at P states: 12-80W `C1`, `C3`, and `C6` transition latency: 2, 10, and 22µs Core static power at `C1`: 1.92-7.11W Core static power at `C3`: 1.64W System bus frequency: 1.2 GHz Superscalar: 5 way Integer/FP ALUs: 3/2 ROB/IQ/LSQ entries: 128/36/72/42 Branch predictor: Bi-mode
Memory Hierarchy	L1I/L1D/L2/L3 size (KB): 64/64/256/4096 L1I/L1D/L2/L3 associativity: 2/4/8/8 DRAM: 8GB/DDR3_1600
Network	NIC: Intel 82574GI Gigabit Link: 10Gbps and 1µs latency
OS	Linux Ubuntu 11.04

To evaluate NCAP, we model a four-node cluster. We configure each simulated node using the parameters tabulated in Table 1 similar to the Intel *i*7-3770 processor. Then we run `Memcached` [27] and `Apache` [28] to get the round trip latency of each request by annotating the source code of `Memcached` and `Apache` clients with `gem5` pseudo instructions. This is not to perturb the system under evaluation by injecting performance-monitoring functions. We modified the source code to implement open-loop `Memcached` and `Apache` clients, and run them on multiple simulated nodes. This is necessary to prevent the (1) client-side queueing bias and (2) dependency between request bursts, which are the two common pitfalls when OLDI benchmarks are evaluated [29]. We set up three clients, each of which sends requests to one server for `Memcached` and `Apache`. With three client nodes, we are able to achieve the maximum load level that an `Apache` or `Memcached` server can sustain without introducing the client-side queuing delay, which was shown to often incur misleading long response time for some requests [29]. Scaling up the number of nodes will distribute requests from more clients to more servers, entailing similar load levels as our setup. To model the bursty nature of the datacenter traffic [30], we set up each client such that it periodically sends a burst of requests (*e.g.*, 200 requests per burst) to the server. Depending on the target load level, we change the period between 1.3 and 20 ms.

We use McPAT [31] to estimate power consumption of processor cores. To estimate the power consumption of processor cores in C states, we make the following assumptions. In C1, C3, and C6 states, processor cores consume no dynamic power, while processor cores consume static power at V used right before transitioning to C1 state, static power at 0.6V, and no static power, respectively.

To demonstrate the advantages of NCAP over a software approach to detect latency-critical requests (*e.g.*, [32]), we also implement the algorithms of `ReqMonitor` and `DecisionEngine` hardware in the NIC device driver. The SoftIRQ interrupt handler for receiving packets calls a `ReqMonitor` function (Figure 5(b)) for each received packet before sending it to upper network layers and increments `ReqCnt` if the packet contains a latency-critical request. We also count the number of transmitted bytes (`TxCnt`) in the SoftIRQ interrupt handler for transmitting packets and utilize a high-resolution kernel timer, which expires every 1ms to determine `ReqRate` and `TxRate`. We transition cores from a C state to the P0 state as soon as we detect a burst of latency-critical requests. The P-state change policy is the same as hardware implementation (Figure 5(c)).

6. Evaluation

In this section, we evaluate the effectiveness of NCAP in reducing both the response time and energy consumption. We consider four power management policies. (1) `perf` disables C states and uses only the `performance` governor. (2) `ond` disables C states and uses only the `ondemand` governor. (3)

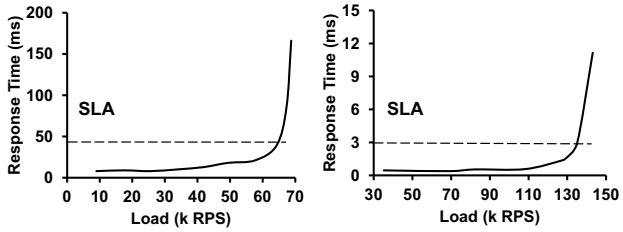

Figure 8: Latency versus load for `Apache` (left) and `Memcached` (right).

`perf.idle` uses both the `performance` and `menu` governors. (4) `ond.idle` uses the `ondemand` and `menu` governors. Then we compare these four policies with three policies of NCAP running atop `ond.idle`. (1) `ncap.sw` is a software-based implementation of NCAP, implementing `ReqMonitor`, `TxBytesCounter`, and `DecisionEngine` in the NIC kernel driver. (2) `ncap.cons` is NCAP with `FCONS` set to 5 to reduce frequency conservatively over five steps. (3) `ncap.aggr` is NCAP with `FCONS` set to 1 to reduces frequency aggressively. We set the threshold values of `DecisionEngine` as follows: `RHT` = 35K requests per second (RPS), `RLT` = 5K RPS, `TLT` = 5M bits per second (BPS), and `CIT` = 500µs, all of which are determined after we analyze the characteristics of `Memcached` and `Apache`.

Servers are often overprovisioned to satisfy a given SLA under a certain high-load level. Therefore, the SLA is typically set near the inflexion point of the latency-load curve [12]. Figure 7 plots the 95th-percentile latency versus load. The measured latency at the inflexion point is 3ms and 41ms for `Memcached` and `Apache`, respectively. We measure the response times of requests and energy consumption after running `Apache` and `Memcached` at low-, medium-, and high-load levels (which corresponds to 24K, 45K, and 66K RPS for `Apache` and 35K, 127K and 138K RPS for `Memcached`). We normalize the measured response times to the SLA [9]. The energy consumption of each policy is normalized to `perf`. We also plot a 200ms snapshot of server's BW(Rx) and F for `ond.idle` and `ncap.cons`. `INT(wake)` in Figure 8 and Figure 9 (right) marks the time that NCAP sends interrupts to the processor cores to proactively transition processor cores from a deep C or P state to the P0 state.

`Apache`. Figure 8 demonstrates that NCAP improves the energy efficiency of the `Apache` server while satisfying the SLA. Analyzing the energy consumption at the low-load level (Figure 8(a)), we observe that `ond` offers 22% lower energy consumption than `perf`. In contrast, `perf.idle` provides 58% lower energy consumption than `perf`. This emphasizes the importance of transitioning cores into a C state when a server is underutilized. Note that `ond.idle` gives marginally (~5%) lower energy consumption than `perf.idle`. This is because `perf.idle` makes cores process incoming requests as fast as possible at the P0 state and then transitions the cores to a deep C state. This is often more energy-efficient than a policy that makes cores process the requests at a deep P state, which consumes lower power but takes a longer time.

While all the seven policies satisfy the SLA at the low-load level (Figure 8(a)), `perf.idle` and `ond.idle` fail to satisfy the SLA at the medium-load level. Therefore, `perf.idle` and `ond.idle` are not viable policies for our server configuration. In contrast, NCAP (`ncap.aggr` and

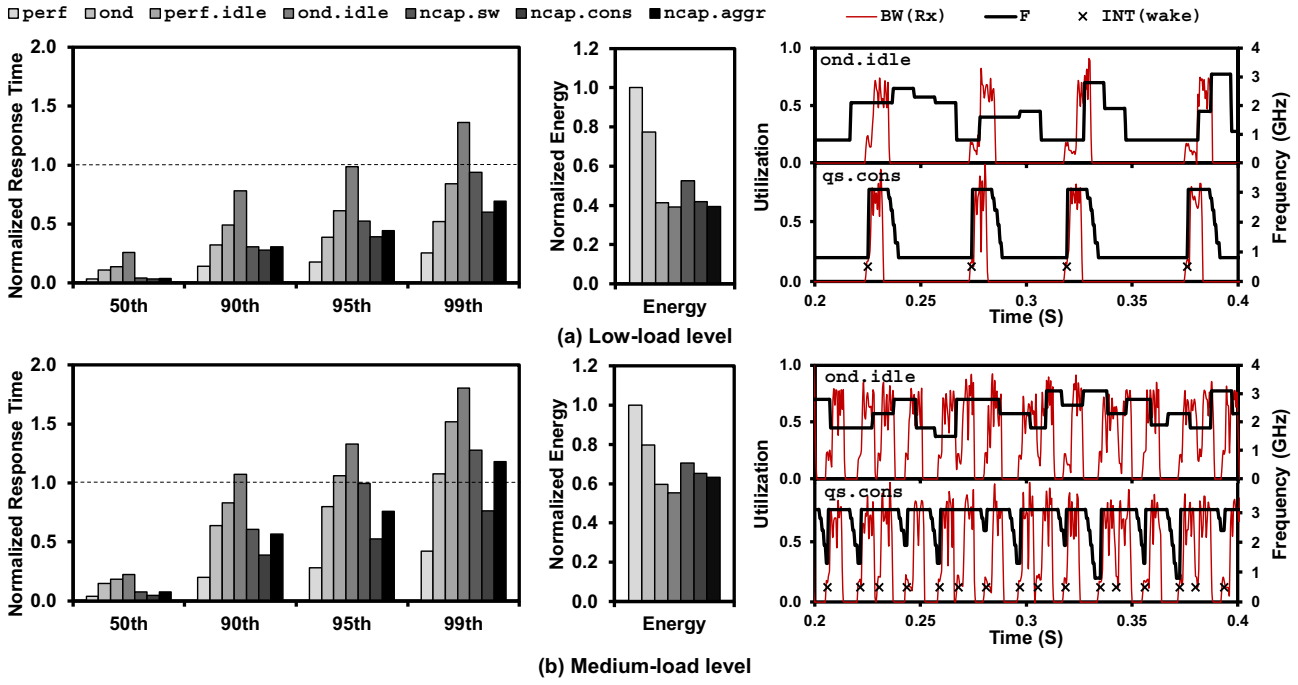

Figure 7: `Apache` response time distribution (left); energy consumption (middle); BW(Rx) vs. F snapshot of `ond.idle` (right top), and `ncap.cons` (right bottom).

ncap.cons) can satisfy the SLA at both the low- and medium-load levels. Amongst conventional policies (*i.e.*, perf, ond, perf.idle, and ond.idle), ond is the most energy-efficient one that can satisfy the SLA. Nonetheless, ncap.aggr offers 49% and 21% lower energy consumption than ond at the low- and medium-load levels, respectively. Compared with ond, the relative energy consumption reduced by NCAP diminishes as the load level increases. This is because the opportunity for cores to run at a deep P state or transition to a deep C state decreases.

A conservative NCAP (ncap.cons) can provide lower response time than an aggressive NCAP (ncap.aggr). This is because ncap.cons can prevent cores from hastily transitioning to a deep P state when the inter-arrival time between BW(Rx) bursts is short. Consequently, the response time disparity between ncap.cons and ncap.aggr is more significant at the medium-load level than the low-load level.

At the low- and medium-load levels, ncap.cons offers 12% and 31% lower 95th-percentile response time than ncap.aggr, but gives 6% and 3% higher energy consumption, respectively. Lastly, the BW(Rx) versus F snapshot in Figure 8(a)-right and (b)-right clearly demonstrates the shortcoming of ond and ond.idle, as described in Section 3, and how NCAP assists ond.idle to quickly handle a sudden processing demand increase for the Apache server.

We observe that ncap.sw can neither fulfil the SLA nor provide significant energy reduction, as the load level increases. ncap.sw gives only 11% lower energy consumption than ond, but 25% higher 95th-percentile response time at the medium-load level. Although this increase in response time does not lead to SLA violations at the given load levels shown in Figure 8, we observe that ncap.sw fails to satisfy the SLA at higher-load levels. This is because at high-load levels, the performance overhead of executing ReqMonitor implemented in software and periodically invoking DecisionEngine is significant and keeps processor cores from spending cycles for processing packets and requests. Even at the low-load level, ncap.aggr and ncap.cons have lower response times than ncap.sw. This demonstrates the effectiveness of proactive power management using an enhanced NIC.

As the NIC and the processor are always highly utilized at high-load levels, idle rarely transitions the processor cores to a C state. Moreover, ond does not change the P state of these cores once ond transitions them to the P0 state. This leaves little opportunity for NCAP to exploit. At such high-load levels, the hardware implementation of NCAP just generates an IT_HIGH interrupt at most every 5ms to set F to the maximum. Note that ReqMonitor and DecisionEngine are not in the critical path and the small interrupt generation rate of NCAP does not incur any notable overhead over the default path for packet processing. Therefore, at a high-load level, the energy consumption and response time of NCAP is identical to perf. Note that the power management policies for servers are aimed to reduce energy consumption at low- and medium-load levels without violating a given SLA for occasional surges in the request rate.

Lastly, depending on a given load level, NCAP implements a race-to-halt policy as it transitions all the processor cores to the P0 state and race to complete the task as quickly as possible, then transition the processor cores to a C state [33]. The dotted boxes in Figure 8(a)-right and (b)-right

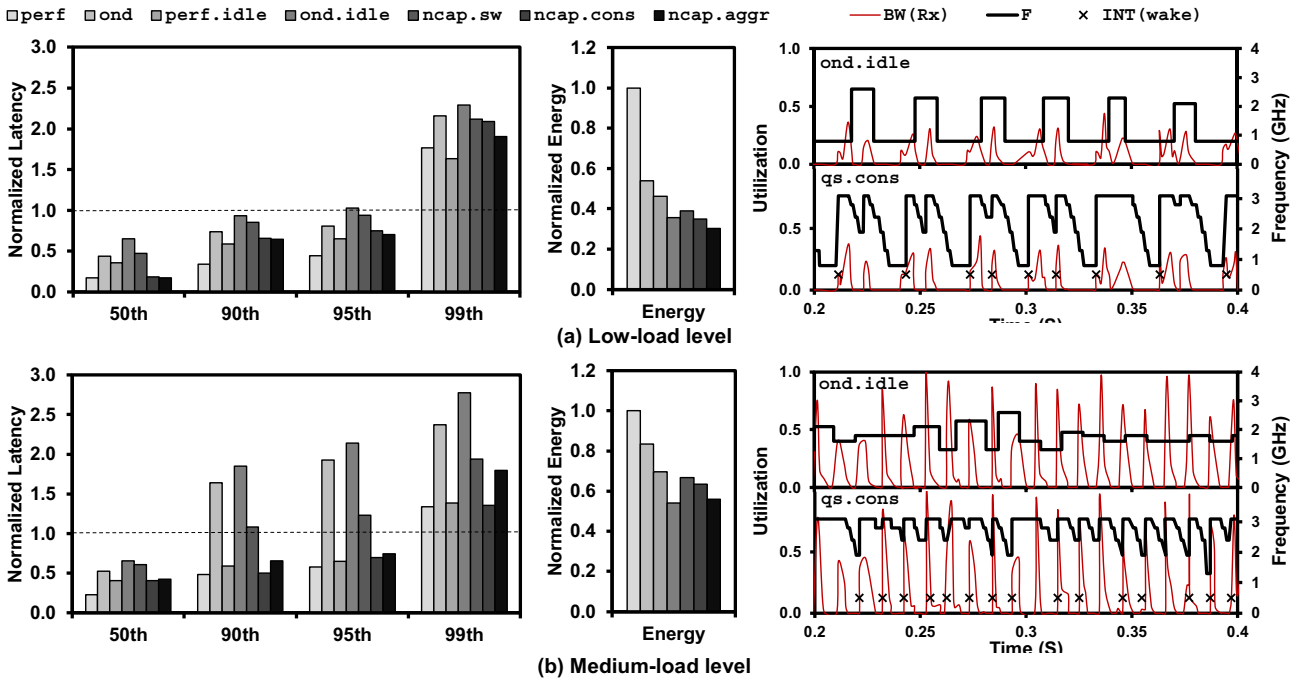

Figure 9: Memcached response time distribution (left); energy consumption (middle); BW(Rx) versus F snapshot of ond.idle (right top), and ncap.cons (right bottom).

clearly show the effectiveness of NCAP in accomplishing this.

Memcached. Figure 9 shows that NCAP improves the energy efficiency of the Memcached server while satisfying the SLA. We observer that the response time of the Memcached server is more sensitive to F than the Apache server. In contrast, the response time of the Apache server was more sensitive to whether or not the menu governor is enabled (perf versus perf.idle) than F (perf versus ond). At the low- and medium-load levels, perf.idle gives 47% and 12% longer 95th-percentile response time than perf, respectively. Moreover, and ond gives 83% and 340% longer 95th-percentile response time than perf at the low- and medium-load levels, respectively. Note that Apache is an I/O-intensive database application that frequently retrieves a large amount of data from a storage device of the server. In contrast, Memcached is a key-value store application that retrieves mostly small values from the main memory of the server [22]. Consequently, Memcached is more sensitive to F than Apache. This is also confirmed by much longer mean response time of Apache (1.7ms) than Memcached (0.6ms).

Although ond cannot react to the surge of BW(Rx) even at the low-load level (Figure 9(a)-right) in time, it can at least identify the high processing demand period incurred by the surge of BW(Rx) and set F to high frequency (between 2.3 to 2.6GHz). However, at the medium-load level, ond fails to detect the high processing demand period and sets F randomly between 1.3 GHz and 2.6GHz. This observation agrees to a previous study demonstrating that considering only CPU utilization for DVFS is often inefficient and leads to SLA violations for servers [12].

Amongst the conventional power management policies, perf.idle is the most energy efficient one that can satisfy the SLA for Memcached at the all load levels. As shown in Figure 9, ncap.cons gives 24% and 9% lower energy consumption, but 15% and 7% longer 95th-percentile response time than perf.idle at the low- and medium-load levels, respectively. In contrast, ncap.aggr gives 34% and 20% lower energy consumption, but 8% and 14% longer 95th-percentile response time than perf.idle at the low- and medium-load levels, respectively.

Increasing the load level reduces the opportunity for NCAP (and other conventional adaptive power management policies such as ond) to transition to a deep P or C state as processor cores are constantly busy. Therefore, the energy consumption of a server deploying NCAP eventually converges to that of perf as the load level increases. Besides, even perf.idle can find little opportunity to transition the processor cores to a C state at high-load levels, leading to no energy reduction compared with perf.

ncap.sw fails to satisfy the SLA for Memcached, as the absolute maximum sustained load level of the Memcached server is 2.1× higher than that of the Apache server (68 RPS versus 143 RPS). This underscores the overhead of ncap.sw for Memcached at high-load levels.

As expected, Figure 9(a)-left shows that the 50th, 90th, and 95th-percentile response times of perf is smaller than perf.idle. However, the 99th-percentile response time of perf.idle is lower than perf. This is because the response time of OLDI applications is a complex function of the interplay amongst several hardware/software components, network traffic patterns and associated queuing delays. We see that enabling/disabling idle often reshapes network traffic patterns. This in turn incurs some unexpected severe network resource contentions between received and transmitted network packets at the beginning of a surge period of BW(Rx), leading to a notable response time increase for a few requests.

7. Discussion

In Section 6, we illustrated the effectiveness of NCAP in improving the energy efficiency of servers running two OLDI applications with inherently different characteristics and QoS requirements at different load levels. A hardware implementation of NCAP significantly reduces the energy consumption of the processor without any SLA violations. Compared with perf, NCAP significantly reduces energy consumption with comparable overall response time. As shown, at both low- and medium-load levels, NCAP exhibit some slack between the achieved 95th-percentile latency and the SLA. This slack can be exploited for further reduction of energy consumption using other techniques [12, 34].

In this paper, we are simulating a cluster with just one OLDI server. However, a production datacenter consists of hundreds or thousands of servers running OLDI applications. One of key characteristics of large-scale datacenters is the load imbalance amongst server nodes. Therefore, there is a significant fraction of underutilized servers even at a high overall load level [12] and NCAP can achieve energy reduction for such underutilized servers.

The gem5 NIC model that we used for our experiments is a single queue model without any TCP offload engines (TOE). NCAP can also be applicable to a server with high-end multi-queue NICs with TOEs. In a multi-queue NIC, as the target core for packet/request processing is known, NCAP changes the P and C states of the target core independent from other cores (per-core versus chip-wide change of P and C states). This can further improve the effectiveness of NCAP. Because TOEs reduce the load on the processors processing packets, a server employing TOE-capable NICs can sustain a higher rate of network packets, compared with a server with a conventional NIC at the same performance state. NCAP can adapt to such scenarios by increasing the threshold values (RHT and RLT in Figure 5(c)). Lastly, as a TOE-enabled NIC holds packets longer time within the NIC than a conventional NIC, NCAP has more slack to hide the latency of processor cores transitioning from a sleep or low-performance state to a high-performance state, which in turn allow NCAP to deploy a more aggressive policy.

8. Related Work

Selective fast performance boost for tail latency. To improve tail latency of OLDI applications, "Adrenaline," query-level performance boosting technique was proposed [32]. Adrenaline identifies latency-critical requests in a network-

stack software layer and rapidly increase V/F using special on-chip voltage regulator (VR) and clock delivery circuits for such requests to reduce tail latency. In contrast, NCAP does not rely on special on-chip VR and clock delivery circuits. Furthermore, NCAP detects latency-critical requests at the lowest network layer (*i.e.*, NIC), which can make much faster detections and decisions than an upper network layer. Finally, NCAP can offer higher energy efficiency as it not only quickly increases performance for latency-critical requests but also proactively decreases performance by observing the rate of transmitted packets and detecting the end of a burst of responses.

Selective performance reduction for energy efficiency. "Pegasus" and "TimeTrader" were proposed to exploit latency slacks of queries that arrive before deadline and reduce energy consumption without increasing SLA violations for OLDI applications [12, 34]. They slow down the system and reshape distribution of latency with the slack. "Rubik" was proposed to find the lowest-performance state of processors that does not violate SLAs using statistical models [35].

Energy-efficiency improvement exploiting sleep states. "PowerNap" was proposed to make server components quickly transition between a high performance state and a sleep state and minimize the idle power consumption of servers [8]. It can effectively reduce idle power consumption for servers with low utilization, but it can experience frequent transitions between the two states and thus increase high-percentile response time for OLDI applications [9]. Thus, it was recommended that low-performance states instead of low-power sleep should be leveraged to improve energy efficiency for OLDI services. Later, it was shown that making processors operate at low-performance states can decrease the energy consumption of servers by up to 15% but increase the high-percentile response time of OLDI applications by 70% [36]. In response, "SleepScale" was proposed to effectively combine various performance and sleep state [10].

Queuing-theoretic analyses of performance and energy. All the aforementioned proposals in this section formulate and tackle server energy-efficiency challenges using queuing-theoretic analysis approaches in which it is challenging to capture the interplay between low-level system hardware and software layers of the computing stack and evaluate schemes modifying such system hardware and software layers. In contrast, NCAP is devised and evaluated based on our full-system simulator that can simulate many nodes connected by a network and run the full system software stack on the simulated system hardware.

9. Conclusion

In this paper, we first made the following three observations. (1) A sudden increase or decrease in network packet rates is highly correlated with the utilization, and thus performance and sleep states of processor cores. (2) The latency-critical requests of OLDI applications are often encapsulated in network packets with a predefined format. (3) The latency to deliver received network packets from a NIC to the processor is notable in network hardware and software layers. Subsequently, based on these three observations, we proposed NCAP, network-driven, packet context-aware power management that enhances a NIC and its driver to assist existing power management policies to improve the energy efficiency of servers running OLDI applications without violating the SLA. More specifically, the enhanced NIC and its driver can detect network packets encapsulating latency-critical requests, speculates the start and completion of a request burst, and predicts the optimal performance and sleep states of processor cores to proactively transition processor cores from a sleep or low-performance state to a high-performance state. We demonstrate the effectiveness of NCAP for two OLDI applications: `Apache` and `Memcached` at various load levels using our enhanced full-system simulator. At low- to medium-load levels, a server deploying NCAP consumes 61~37% lower processor energy than the baseline while satisfying a given SLA. Furthermore, NCAP can provide notably lower energy consumption with faster 95th-percentile response time than an approach that detects and reacts to latency-critical requests in a network software layer.

Acknowledgement

This work is supported in part by grants from NSF (CNS-1600669 and CNS-1557244), and NCSA Faculty Fellow Program. Nam Sung Kim and Daehoon Kim are the co-corresponding authors. Nam Sung Kim has a financial interest in AMD and Samsung Electronics.

References

[1] J. Dean and L.A. Barroso, "The Tail at Scale," Communications of the ACM*,* vol. 56, no. 2, 2013.

[2] P. Delforge, 2015. [Online]. Available: https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy.

[3] L.A. Barroso, J. Clidaras and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines," Second Edition, Morgan & Claypool, 2013.

[4] V. Pallipadi and A. Starikovskiy, "The ondemand Governor: Past, Present, and Future," in Linux Symposium, 2006.

[5] V. Pallipadi, S. Li and A. Belay, "cpuidle: Do nothing, efficiently," in Linux Symposium, 2007.

[6] [Online]. Available: http://www.acpi.info/.

[7] 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 (ATC), 2014.

[8] D. Meisner, B.T. Gold and T.F. Wenisch, "PowerNap: Eliminating Server Idle Power," in International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2009.

[9] D. Meisner, C.M. Sadler, L.A. Barroso, W.-D. Weber and T.F. Wenisch, "Power Management of On-Line Data-Intensive Services," in International Symposium on Computer Architecture (ISCA), 2011.

[10] 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 Architecture (ISCA), 2014.

[11] M. Flajslik and M. Rosenblum, “Network Interface Design for Low Latency Request-Response Protocols,” in USENIX Annual Technical Conference (ATC), 2013.

[12] D. Lo, L. Cheng, R. Govindaraju, L. Barroso and C. Kozyrakis, “Towards Energy Proportionality for Large-Scale Latency-Critical Workloads,” in International Symposium on Computer Architecture (ISCA), 2014.

[13] Intel Corporation, [Online]. Available: www.intel.com/assets/PDF/designguide/321736.pdf .

[14] A. Bashir, J. Li, K. Ivatury, N. Khan, N. Gala, N. Familia and Z. Mohammed, “Fast Lock Scheme for Phase-Locked Loops," in Custom Integrated Circuits Conference (CICC), 2009.

[15] D. Brodowski, “CPU Frequency and Voltage Scaling Code in the Linux kernel,” [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[16] “VRM and EVRD 11.1 Design Guidelines,” [Online]. Available: http://www.intel.com/content/www/us/en/power-management/voltage-regulator-module-enterprise-voltage-regulator-down-11-1-guidelines.html.

[17] [Online]. Available: http://www.infinibandta.org/.

[18] M. Rashti and A. Afsahi, "10-Gigabit iWARP Ethernet: Comparative Performance Analysis with InfiniBand and Myrinet-10G," in Parallel and Distributed Processing Symposium (IPDPS), 2007.

[19] S. Larsen, P. Sarangam, R. Huggahalli and S. Kulkarni, “Architectural Breakdown of End-to-End Latency in a TCP/IP Network," International Journal of Parallel Programming, vol. 37, no. 6, pp. 556--571, 2009.

[20] “Interrupt Moderation Using Inter GbE Contorllers,” [Online]. Available: http://www.intel.my/content/dam/doc/application-note/gbe-controllers-interrupt-moderation-appl-note.pdf.

[21] M. Alian, D. Kim and N.S. Kim, “pd-gem5: Simulation Infrastructure for Parallel/Distributed Computer Systems,” IEEE Computer Architecture Letters, vol. 15, no. 1, 2016.

[22] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang and M. Paleczny, “Workload Analysis of a Large-Scale Key-Value Store,” SIGMETRICS Performance Evaluation Review, vol. 40, no. 1, 2012.

[23] “Hypertext Transfer Protocol,” [Online]. Available: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

[24] P. Mochel, “The sysfs File System,” in Linux Symposium, 2005.

[25] “MTU,” [Online]. Available: https://en.wikipedia.org/wiki/Maximum_transmission_unit.

[26] S. Vasileios, A. Bagdia, A. Hansson, P. Aldworth and K. Stefanos, “Introducing DVFS-Management in a Full-System Simulator," in International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2013.

[27] 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 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

[28] “ab - Apache HTTP Server Benchmarking Tool,” [Online]. Available: https://httpd.apache.org/docs/2.2/programs/ab.html.

[29] Y. Zhang, D. Meisner, J. Mars and L. Tang, “Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference,” in International Symposium on Computer Architecture (ISCA), 2016.

[30] T. Benson, A. Akella and D.A. Maltz, “Network Traffic Characteristics of Data Centers in the Wild," in SIGCOMM Conference on Internet Measurement (IMC), 2010.

[31] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen and N.P. Jouppi, “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures,” in International Symposium on Microarchitecture (MICRO), 2009.

[32] C.-H. Hsu, Y. Zhang, M.A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang and R.G. Dreslinski, “Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting,” in International Symposium on High-Performance Computer Architecture (HPCA), 2015.

[33] S. Dawson-Haggerty, A. Krioukov, D. Culler “Power Optimization - A Reality Check,” [Online]. Available: https://people.eecs.berkeley.edu/~krioukov/realityCheck.pdf.

[34] B. Vamanan, H. Sohail, J. Hasan and T. Vijaykumar, "TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search," in International Symposium on Microarchitecture (MICRO), 2015.

[35] 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.

[36] F. Rossi and M. Conterato, "Evaluating the Trade-off between DVFS Energy-Savings and Virtual Networks Performance," in Conference on Information-Centric Networking (ICN), 2014.

[37] T. Benson, A. Akella and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in SIGCOMM Conference on Internet Measurement (IMC), 2010.

[38] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna and S. Sardashti, “The gem5 Simulator,” SIGARCH Computer Architecture News, vol. 39, no. 2, 2011.