

vSPACE: Supporting Parallel Network Packet Processing in Virtualized Environments through Dynamic Core Management

Gyeongseo Park* DGIST/ETRI Republic of Korea gspark@dgist.ac.kr

Yunhyeong Jeon DGIST Republic of Korea yhjeon@dgist.ac.kr Minho Kim DGIST Republic of Korea mhkim@dgist.ac.kr

Sungju Kim DGIST Republic of Korea sungju_kim@dgist.ac.kr

Daehoon Kim[†] Yonsei University Republic of Korea daehoonkim@yonsei.ac.kr Ki-Dong Kang ETRI Republic of Korea kd_kang@etri.re.kr

Hyosang Kim DGIST Republic of Korea hyosangkim@dgist.ac.kr

Abstract

Data centers face significant performance challenges with parallel processing for network I/O in virtualized environments, particularly for latency-critical (LC) workloads that must satisfy strict Service Level Objectives (SLOs). While previous studies have addressed performance challenges in network I/O virtualization, they overlook the impact of excessive parallelism on the performance of Virtual Machines (VMs). We observe that excessive parallelization for VMs and network I/O processing can lead to core oversubscription, resulting in significant resource contention, frequent preemptions, and task migrations. Based on these observations, we propose vSPACE, dynamic core management specifically designed to support parallel network I/O processing in virtualized environments efficiently. To reduce scheduling contention, vSPACE creates distinct core allocation groups for VM and network I/O and assigns dedicated cores to each. Then, it dynamically adjusts the number of allocated cores to enforce appropriate parallelism for VMs and network I/O processing based on varying demands. vSPACE employs continuous monitoring and a heuristic algorithm to periodically determine appropriate core allocation, addressing excessive contention and improving energy and resource efficiency. vSPACE operates in three modes: performance improvement, energy efficiency, and resource efficiency. Our evaluations demonstrate that vSPACE significantly enhances throughput by up to 4.2× compared to existing core allocation approaches and improves energy and resource efficiency by up to 16.5% and 30.5%, respectively.

[†]Daehoon Kim is the corresponding author.



This work is licensed under a Creative Commons Attribution International 4.0 License.

PACT '24, October 14–16, 2024, Long Beach, CA, USA © 2024 Copyright held by the owner/author(s). ACM ISBN 979-8-4007-0631-8/24/10 https://doi.org/10.1145/3656019.3689610

ACM Reference Format:

Gyeongseo Park, Minho Kim, Ki-Dong Kang, Yunhyeong Jeon, Sungju Kim, Hyosang Kim, and Daehoon Kim. 2024. vSPACE: Supporting Parallel Network Packet Processing in Virtualized Environments through Dynamic Core Management. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3656019.3689610

1 Introduction

Efficiently exploiting the parallelism of multi-core processors is crucial for the performance of virtualized systems, which typically consolidate multiple virtual machines (VMs). Each Virtual Machine (VM) not only utilizes multiple vCPUs with multithreading capabilities but also applies parallel processing to network I/O handling with its own virtual Network Interface Cards (vNICs) [2, 13]. While parallel processing can enhance performance by distributing tasks across multiple cores, it introduces significant performance challenges, particularly in network I/O virtualization. The inherent parallelism in both VM operations and network I/O processing often leads to the oversubscription of processor cores, increasing scheduling overheads. This excessive parallelism intensifies existing virtualization challenges, including increased latency from unscheduled vCPUs, core resources competition from multiple network I/O processing demands (e.g., physical and virtual Network Interface Cards (pNICs and vNICs)), frequent VM exits from virtual interrupts [11, 14, 15, 18, 19, 33]. These issues are problematic for Latency-Critical (LC) workloads, where satisfying Service Level Objectives (SLOs) is essential.

Although parallel processing offers opportunities to improve performance, previous studies with hardware and software solutions have primarily focused on addressing I/O virtualization overhead within limited parallelism and capabilities. SR-IOV [12], a hardware solution, reduces I/O overhead by allowing VMs direct access to devices, enhancing performance. However, SR-IOV supports limited parallelism as it relies on a static level of parallelism. Additionally, deploying SR-IOV for advanced functionalities (e.g., overlay networks) demands specific devices with particular capabilities (e.g.,

^{*}Work was done while at DGIST

VXLAN), thus increasing deployment complexity in multifaceted settings. Consequently, SR-IOV often remains an optional feature in cloud infrastructures due to these limitations. On the software side, hBalance/vBalance[8, 10] aim to mitigate I/O processing delays within vCPUs by redirecting I/O traffic from inactive to active vCPUs. ES2[15] introduces an adaptive hybrid I/O handling model that alternates between exit-based notifications (i.e., virtual interrupts) during low load periods and a non-exit polling mode under higher load conditions. These methods retain the adaptability and scalability of software-based I/O virtualization, allowing dynamic adjustments to workload changes without hardware reconfiguration. However, their focus on single I/O processing overlooks the broader potential advantages of parallel processing for multiple tasks such as VM and I/O.

Excessive parallelization for VMs and network I/O leads to core oversubscription, causing inefficient context switches and task migrations. Our investigation uncovers that network I/O virtualization amplifies the challenges due to its involvement in a wide range of multiple tasks, including parallel packet processing on pNIC and vNICs, and executing workloads on multiple vCPUs. This oversubscription presents challenges in balancing core allocation between vCPUs and network I/O processing.

To overcome these challenges, we introduce vSPACE, a sophisticated software-centric core management technique designed for parallel network I/O processing in virtualized environments, significantly enhancing the performance and efficiency of LC workloads. We observe that time sharing the cores for both vCPUs and network I/O processing results in considerable contention, which can be effectively addressed by dedicating separate cores to each. Based on the observation, vSPACE creates distinct core allocation groups for vCPUs and network I/O parallel processing, assigning dedicated cores to each group. Then, vSPACE dynamically adjusts the number of cores dedicated to vCPUs and network I/O processing based on fluctuating demands, preventing core utilization saturation. Through continuous monitoring for each group and a heuristic algorithm, vSPACE periodically decides appropriate core allocation, addressing excessive contention and improving energy and resource efficiency. vSPACE introduces three operational modes, vSPACE-P for performance, vSPACE-E for energy efficiency, and vSPACE-R for resource efficiency, each designed to meet specific requirements while enhancing overall system performance and efficiency.

To the best of our knowledge, this study is the first comprehensive dynamic core management approach considering both vCPUs and network I/O processing in I/O virtualization environments. While parallel processing for VMs and network I/O increases resource contention, vSPACE addresses this with isolation and dynamic core allocation, enhancing performance, energy efficiency, and resource utilization. vSPACE-P significantly enhances the maximum Query Per Second (QPS), achieving up to a 4.2 × improvement over static core allocation methods. vSPACE-E reduces energy consumption by up to 16.5% compared to state-of-the-art dynamic core allocation techniques focused on energy efficiency [28]. Furthermore, vSPACE-R leads to more effective core allocation for VMs running Best Effort (BE) workloads, showing up to a 30.5% improvement over dynamic core co-allocation study for resource efficiency [34]. This comprehensive approach not only outperforms existing scheduling and core allocation methods but also ensures

cloud deployment adaptability without requiring modifications to the hypervisor, guest OS, or hardware. The primary contributions of this work include:

- We emphasize the significance of core allocation for packet processing in virtualized environments, advocating for equal consideration with vCPU, diverging from traditional views.
- We identify a performance bottleneck in network packet processing from core scheduling contention, showing that dynamic and isolated core allocation can overcome it.
- vSPACE methodology introduces two novel strategies: distinct and dynamic core allocation for vCPUs and packet processing.
- The three vSPACE modes outperform state-of-the-art scheduling and core allocation approaches in performance, energy, and resource efficiency, respectively.

2 Background

High-performance network technologies in modern pNICs and their operating systems support parallel packet processing across multi-core processors [23]. This is facilitated by techniques like Receive Side Scaling (RSS) for incoming traffic and Transmit Packet Steering (XPS) for outgoing traffic [35], allowing efficient packet distribution to multiple processor cores. To enable efficient distribution of network packets across multiple pCPUs, pNICs provide multiple physical network queues (pNOs), each of which is mapped to an individual pCPU. The packet distribution relies on hashing IP addresses and port numbers, ensuring packets from the same connection are handled by the same core. Virtualized environments extend these principles through vNIC employing that use virtual network queues (vNQs) linked to vCPUs, facilitating network traffic management [36]. There are two components that process packets with the vNICs: a back-end device and a front-end driver. The backend device generally emulates the hardware functionality of NICs on the hypervisor using hypervisor-level threads, such as vNQs and registers, which involves memory copying to transfer packets from pNQs to vNQs, and vice versa. The front-end driver provides interfaces that enable guest VMs to manipulate back-end devices. When network packets arrive or are transmitted, the corresponding vCPUs linked to vNQs are alerted through virtual interrupt requests (vIRQs). This process ensures that both the hypervisor-level threads simulating the vNQs and the vCPUs tied to these queues are simultaneously activated for parallel packet processing.

Figure 1 shows the overview of parallel packet processing in a virtualized environment. When a packet arrives at the pNIC, the packet is delivered to a pNQ via the RSS technique. The packet is copied to a software Rx queue in the hypervisor, and a pIRQ is sent to the pCPU corresponding to the pNQ, informing the packet's arrival (①). The pCPU immediately schedules the pIRQ handler, processing the packet and delivering the packet to a Received (Rx) vNQ (②). The back-end device then sends a vIRQ to the vCPU assigned to the vNQ (③). The vCPU executes the vIRQ handler of the front-end driver within the VM, and the front-end driver delivers the packet's payload to the application running within the VM (④). For packet transmission, the application within the VM first packetizes the data through the network stack (④). The front-end driver enqueues the packet in a Tx vNQ and triggers a VM exit to notify the I/O event for the hypervisor (④). The back-end device



Figure 1: The overview of network packet processing in a virtualized environment.

transfers the packet to the pNIC driver (③), and the driver copies the packet to a Tx pNQ (④). After transmission, the pNIC/vNIC sends a pIRQ/vIRQ to the pCPU/vCPU assigned to the pNQ/vNQ, and the pCPU/vCPU preemptively schedules the pIRQ/vIRQ handler to complete the packet transmission process (⑤/⑥).

The number of pNQs/vNQs of pNICs/vNICs determines the degree of parallelism for parallel packet processing. In other words, the number of pNQs/vNQs establishes the degree to which pIRQs/vIRQs can be sent in parallel, subsequently activating pCPUs/vCPUs equal to the number of pNQs/vNQs. Furthermore, the back-end devices, which are typically composed of kernel threads on the hypervisor, are also activated, corresponding to the number of vNQs.

3 Impact of Parallel Packet Processing with I/O Virtualization

This section examines how strategic core management influences LC workload performance within I/O virtualization. Our focus lies on two critical areas: the adjustment of pNQs and vNQs, which affects the engagement of pCPUs and vCPUs in packet processing, and the impact of core management policies on overall system performance. Specifically, adjusting the number of pNQs in pNICs and vNQs in vNICs changes network I/O parallelism levels; high levels of parallelism intensify oversubscription. We scrutinize core management policies, encompassing modifications in pNQs, vNQs, vCPUs numbers, and their corresponding pCPU allocations. For our experiments, we employ memcached, a key-value store application running on a VM with 20 threads across 20 vCPUs and pCPUs, generating network traffic using mutilate based on Facebook's ETC trace [4]. We conduct over 10 trials while we plot the mean with shaded regions and error bars representing 95% confidence intervals.

3.1 Impact of Parallel Packet Processing on LC Workload

We investigate the impact of parallel packet processing on tail response latency for LC workloads by adjusting the number of pNQs



Figure 2: 95th percentile latency of memcached as the number of pNQs and vNQs changes.



Figure 3: System events across all cores in memcached at 50KQPS as the number of pNQs and vNQs changes.

and vNQs. These configurations are denoted as *n*pNQ-*n*vNQ, where *n* indicates the number of network queues. In Figure 2, we examine the P95 against Query Per Second (QPS) for configurations ranging from minimal to maximal network queues: 1pNQ-1vNQ, 1pNQ-20vNQ, 20pNQ-1vNQ, and 20pNQ-20vNQ. Our findings highlight a deterioration in performance with maximized vNQs or pNQs compared to the baseline 1pNQ-1vNQ. For example, 1pNQ-1vNQ violates the SLO from 110KQPS while 1pNQ-20vNQ and 20pNQ-1vNQ violate the SLO from 70KQPS and 110KQPS, respectively. Particularly, 1pNQ-20vNQ suffers from further performance degradation, as having multiple vNQs induces more scheduling overhead than multiple pNQs; we delve into details later. Interestingly, even the 20pNQ-20vNQ, which maximizes both pNQs and vNQs, shows worse performance at several load levels (from 80 KQPS to 120 KQPS) compared to 1pNQ-1vNQ. However, the 1pNQ-1vNQ configuration experiences a sharp increase in tail latency around 120KQPS due to nearly 100% CPU utilization. Conversely, the 20pNQ-20vNQ configuration mitigates CPU saturation with parallel packet processing but incurs scheduling overhead. The results indicate the need to balance core allocation for NQs to reduce scheduling contention and prevent CPU saturation.

Figure 3 presents system events that reveal the scheduling overheads at 50KQPS where all configurations satisfy the SLO. Rising pNQs and vNQs directly correlate with an increase in pIRQs and vIRQs, prompting more frequent context switches and migrations. For example, the 20pNQ-20vNQ setup sees pIRQs and vIRQs increase by 1.9x and 1.6x, respectively, compared to 1pNQ-1vNQ, resulting in a similar increase in context switches. The competitive environment significantly increases migrations by up to 466.5×, as



Figure 4: 95th percentile latency of memcached as the number of pNQs, vNQs, and vCPUs changes.

the hypervisor scheduler reallocates vCPUs and vNQs across pC-PUs. Moreover, vNQs significantly increase scheduling overheads more than pNQs. For instance, 1pNQ-20vNQ increases the number of context-switches and migration events by 2.0× and 288.2×, respectively, compared to 20pNQ-1vNQ. This is because vNQs engage both the corresponding vCPUs and the hypervisor threads that emulate them, leading to higher competition for pCPU resources. In contrast, pNQs trigger packet processing directly via pIRQs, avoiding hypervisor scheduling. Furthermore, hypervisor-level threads for vNQs require substantial computational resources to perform memory copying for packet delivery between vNQs and pNQs, whereas pNICs offload packet copying to Direct Memory Access (DMA) engines, reducing their resource demands. Consequently, configurations like 20pNQ-20vNQ underperform compared to 1pNQ-1vNQ, despite spreading pCPU and vCPU utilization.

3.2 Impact of Core Allocation and Parallel Packet Processing on LC Workload

To mitigate the scheduling overheads associated with parallel network packet processing, we apply two intuitive policies. Firstly, we distinctly allocate pCPUs for vCPUs and NQs (i.e., both pNQs and vNQs), to reduce resource contention. Secondly, we adjust the allocation of pCPUs to vCPUs and NQs to ensure their combined count aligns with the total number of available pCPUs, effectively preventing resource overcommitment. We evaluate two primary strategies in virtualized environments: Time-Sharing and Isolation. Time-Sharing involves vCPUs and NQs sharing pCPUs equally. For instance, with 20 total pCPUs, the combined count of vCPUs, pNQs, and vNQs also totals 20, ensuring a uniform distribution of resources across all tasks. Conversely, the Isolation strategy assigns network queues and vCPUs to separate pCPUs, aligning each parallelism level with the allocated pCPUs. This configuration is denoted as nvCPU-nNQ, where n indicates the number of allocated pCPUs for both vCPUs and NQs, ensuring the total number of allocated resources matches the number of pCPUs.

Our results, depicted in Figure 4, demonstrate how different strategies influence the P95. Isolation outperforms Time-Sharing in satisfying the SLO, particularly at a higher query rate. Almost Isolation configurations satisfy the SLO at 250KQPS, whereas Time-Sharing violate The SLO beyond 150K QPS. Furthermore, the effectiveness of Isolation emphasizes the critical need for carefully balanced core allocation for NQs and vCPUs. Insufficient core allocation can lead to excessive concentration on either NQs



Figure 5: Comparison of system events in memcached between Time-Sharing and Isolation.



Figure 6: 95th percentile latency for Time-Sharing and Isolation based on vCPUs/NQs adjustments.

or vCPUs (approaching 100% core utilization), resulting in performance degradation as seen in the 18vCPU-2NQ and 8vCPU-12NQ. This can adversely impact latency to escalate exponentially, sometimes resulting in outcomes worse than those of Time-Sharing.

Figure 5 provides a detailed comparison of system events for both Time-Sharing and Isolation (*i.e.*, 16vCPU-4NQ), illustrating the areas of improvement. Isolation can avoid core oversubscription with separated cores and reduced parallelism level, resulting in fewer pIRQs and vIRQs, thereby mitigating preemptions and VM exits. Compared to Time-Sharing, Isolation decreases the count of pIRQs and vIRQs by up to 51.7% and 15.1%, respectively. Moreover, Isolation reduces context switches and migrations by up to 58.6% and 99.6%, respectively.

Additionally, we delve into performance improvements achieved by isolation and parallelism management. Figure 6 shows the 95th percentile latency across KQPS levels for four distinct configurations, employing either Time-Sharing or Isolation, with varying parallelism levels for VM and I/O. The baseline configuration, 20vCPU-20vNQ(w/o iso.), shows the least effective performance. In contrast, 16vCPU-4NQ(w/o iso.), which involves parallelism management for VM and I/O without Isolation, shows improved performance. This improvement is primarily due to reduced contention for pCPUs, indicating that reducing parallelism levels to avoid overcommitment can improve latency. Furthermore,



Figure 7: vSPACE architecture.

20vCPU-20vNQ(w/ iso.), which uses Isolation without changing parallelism levels for I/O and VM (both at 20), shows performance improvement. This indicates that Isolation alone, even without changing the number of vCPUs or NQs, can effectively enhance system performance. The 16vCPU-4NQ(w/ iso.) shows the best performance by combining these two strategies.

Architecture 4

4.1 Overview

We propose vSPACE that, in virtualized environments, efficiently Supports PArallel network paCkEt processing through dynamic core management for vCPUs and NQs for LC workloads. Based on our observation, vSPACE dynamically allocates pCPUs to vCPUs and NQs (i.e., pNQs and vNQs), ensuring the satisfaction of the target SLO. Adjustments in pCPU allocation for vCPUs and NQs directly influence the configuration of vCPUs and NQs, aligning resources with demand. For instance, if vSPACE designates one pCPU for NQ (i.e., 1NQ) and two pCPUs for vCPUs (i.e., 2vCPU), resulting in corresponding adjustments in the numbers of pNQs, vNQs, and vCPUs to one, one, and two, respectively. In this configuration, the single pNQ and vNQ share the allocated pCPU.

The architecture of vSPACE is depicted in Figure 7, consisting of two main components: vSPACE-Host and vSPACE-Guest. vSPACE-Host, operating at the hypervisor level, periodically measures the pCPU utilization by vCPUs and NQs, along with the tail response latency. Utilizing this data, vSPACE-Host searches for the most efficient core allocation configuration. To achieve precise core allocation, vSPACE-Host applies two critical utilization thresholds for vCPUs and NQs. These thresholds represent the maximum allowable pCPU utilization for vCPUs and NQs to satisfy the SLO. The adjustment involves scaling the number of pNQs and allocating pCPU for both vCPUs and NQs. Concurrently, vSPACE-Guest modifies the number of active vNQs and vCPUs within the VM. vSPACE offers three modes: performance mode (i.e., vSPACE-P), energy-efficiency mode (i.e, vSPACE-E), and resource-efficiency mode (i.e., vSPACE-R). In vSPACE-P, all pCPUs are fully activated to maximize performance and reduce latency, even when fewer cores can satisfy the SLO, but this mode consumes more energy. vSPACE-E reduces energy consumption by dynamically allocating pCPUs to vCPUs and NQs

A	lgorith	m 1	: D	ynamic	Core	Alloca	tion	of	vSPAC	CE-F	2
---	---------	-----	-----	--------	------	--------	------	----	-------	------	---

Require: TotalUtil_{vCPU}, TotalUtil_{NO} (Measured metrics)

- 1: TotalCore = the total number of pCPUs
- 2: $ReqCore_{vCPU} = TotalUtil_{vCPU}/Th_{vCPU}$ 3: $ReqCore_{NQ} = TotalUtil_{NQ}/Th_{NQ}$
- 4: ReqCore_{total} = ReqCore_{vCPU} + ReqCore_{NQ} 5: NextCore_{vCPU} = round(TotalCore * ReqCore_{vCPU}/ReqCore_{total})
- 6: $NextCore_{NQ} = round(TotalCore * ReqCore_{NQ}/ReqCore_{total})$

as the load changes, maintaining the SLO but resulting in higher latency than vSPACE-P. vSPACE-R enhances resource efficiency by allocating idle pCPUs to a VM running a CPU-intensive workload (Best Effort) but causing lower throughput in LC workloads compared to vSPACE-P and vSPACE-E due to contention from the co-running workload.

Dynamic Core Allocation 4.2

vSPACE dynamically manages core allocation for vCPUs and NQs by monitoring pCPU utilization to enhance performance and efficiency. The primary objective is to prevent pCPU resource saturation for vCPUs and NQs, using established utilization thresholds. These thresholds are set to identify potential pCPU overload conditions that have a risk of SLO violation.

Algorithm 1 outlines the operation of vSPACE-P. vSPACE-P utilizes all available pCPUs while dynamically adjusting the proportion of pCPUs allocated to vCPUs and NOs. The algorithm begins with identifying the number of pCPUs demanded for vCPUs and NQs. It calculates the required pCPUs by dividing their respective total pCPU utilizations by their individual thresholds and rounding to the nearest integer (lines 2-3). The algorithm then calculates the total number of required pCPUs by summing the determined pCPUs for vCPUs and NQs (line 4). The subsequent pCPU allocation to vCPUs and NQs is determined by proportionally distributing the maximum number of pCPUs relative to their required pCPUs (lines 5-6). This approach ensures efficient utilization of all pCPUs while maintaining a fair distribution between vCPUs and NQs.

Algorithm 2 describes vSPACE-E's dynamic core allocation strategy, utilizing a MARGIN to modulate core adjustments. Higher MARGIN values promote stability by resisting rapid changes in allocated pCPUs, whereas lower values favor energy efficiency through more aggressive core reductions. It calculates the number of pCPUs demanded for both vCPUs and NQs, similar to vSPACE-P (lines 2-3). The algorithm starts by summing the pCPUs required by vCPUs and NQs, then verifying if the summation result exceeds the total number of pCPUs (lines 4-5). If so, the algorithm employs a similar method as vSPACE-P to determine the core allocation for vCPUs and NQs, where the pCPUs are allocated based on their respective requirements (lines 6-7). However, if the required pCPUs do not exceed the total number of pCPUs, the algorithm compares the previously allocated pCPUs with the current requirements for the vCPUs and NQs (lines 9 and 14). If the previous core allocation is larger than the current requirements (i.e., the algorithm determines that a decrease in allocated cores is necessary), the next core allocation is determined with increased total pCPU utilization by the margin multiplier (lines 10 and 15). It enhances the stability of core allocation by making it more difficult to decrease the number of pCPUs. Conversely, if the previous core allocation is smaller than the current requirements (i.e., the algorithm determines that

A	lgorithm	2:	Dynamic	Core	Allocation	of	vSPA	CE-I	E
---	----------	----	---------	------	------------	----	------	------	---

Require: MARGIN (A margin to stabilize core allocation) Require: PrevCorevCPU, PrevCoreNQ (Previous core allocation) **Require:** $TotalUtil_{vCPU}$, $TotalUtil_{NQ}$ (Measured metrics) 1: TotalCore = the total number of pCPUs 2: $ReqCore_{vCPU} = ceil(TotalUtil_{vCPU}/Th_{vCPU})$ 3: $ReqCore_{NQ} = ceil(TotalUtil_{NQ}/Th_{NQ})$ $ReqCore_{total} = ReqCore_{vCPU} + ReqCore_{NQ}$ 4: 5: if *ReqCore*total > *TotalCore* then 6: NextCore_{vCPU} = round(TotalCore * ReqCore_{vCPU}/ReqCore_{total}) 7: NextCore_{NQ} = round(TotalCore * ReqCore_{NQ}/ReqCore_{total}) 8: else if $PrevCore_{vCPU} > ReqCore_{vCPU}$ then 9: $NextCore_{vCPU} = ceil(MARGIN * TotalUtil_{vCPU}/Th_{vCPU})$ 10: 11: else 12: NextCore_{vCPU} = ReqCore_{vCPU} 13: end if 14: if $PrevCore_{NO} > ReqCore_{NO}$ then $NextCore_{NQ} = ceil(MARGIN * TotalUtil_{NQ}/Th_{NQ})$ 15: else 16: $NextCore_{NQ} = ReqCore_{NQ}$ 17: end if 18: 19: end if

an increase in allocated cores is necessary), the required pCPUs are directly assigned as the next core allocation (lines 12 and 17).

vSPACE-R adopts a similar strategy to vSPACE-E for allocating cores but adds a step to enhance resource efficiency. It allocates idle pCPUs to VMs running BE workloads, after allocating pCPUs to vCPUs and NQs for VM executing LC workload. If no pCPUs are idle, the BE VM is temporarily suspended until pCPUs become idle.

4.3 Exploration of Thresholds

We introduce a threshold exploration policy, preventing both underutilization (by avoiding excessively low thresholds) and overload risks (by avoiding excessively high thresholds). The policy gradually adjusts thresholds based on the observed utilization of vCPUs and NQs. It changes thresholds for either vCPUs or NQs, with a preference for maintaining higher thresholds to ensure efficient resource utilization while satisfying the SLO.

Outlined in Algorithm 3, the process requires three constants: AllowVio (allowable SLO violation percentage), Reward (reward factor for non-violation), and Weight (weight for threshold updates). Higher Reward and Weight lead to quicker adjustments, while lower values ensure stability. Periodically, it collects key performance indicators, P95 latency and average CPU utilization for both vCPUs $(util_{vCPU})$ and NQs $(util_{NQ})$, aggregating these into a dataset S (lines 1-4) Once a notification is received, the algorithm initiates the process for determining new thresholds (line 5). This involves first retrieving the most recent thresholds for vCPUs (TH_{vCPU}) and NQs (TH_{NO}) and then initializing corresponding temporary thresholds (TH'_{vCPU}, TH'_{NO}) (lines 6-7). The algorithm filters the SLO-violating samples into S_{vio} (line 8). The algorithm calculates allowVioCount, which represents the maximum number of violations allowable, based on AllowVio and the total number of samples in set *S*, and then determines the objective count to be decreased to satisfy the SLO (decCount) (lines 9-10). If decCount exceeds zero, indicating the need for threshold adjustments to satisfy the SLO, the algorithm embarks on a process to establish new thresholds (line 10). It sorts in descending order and stores Svio into S_{vCPU} and S_{NQ} based on their respective utilizations ($util_{vCPU}$ and

Algorithm 3: Threshold Exploring Algorithm

Require: AllowVio = allowable SLO violation percentage $(0 \le AllowVio \le 1)$ **Require:** Reward = reward for non-violation (Reward ≥ 1) **Require:** Weight = weight for threshold update $(0 \le Weight \le 1)$ 1: if receives P95, $util_{vCPU}$, $util_{NQ}$ then $s = [P95, util_{vCPU}, util_{NQ}]$ 2: 3: Append s to S 4: end if 5: if notified of threshold exploration then TH_{vCPU} , TH_{NQ} = last thresholds for vCPUs and NQs 6: $TH'_{vCPU}, TH'_{NQ} = Th_{vCPU}, Th_{NQ}$ S_{vio} = filter SLO-violating from S7: 8: $allowVioCount = ceil(count(S) \times AllowVio)$ 9: $decCount = count(S_{vio}) - allowVioCount$ 10: 11: if decCount > 0 then $S_{vCPU}, S_{NQ} =$ Descending Sort S_{vio} by $util_{vCPU}, util_{NQ}$ 12: 13: while decCount > 0 do s_{vCPU}, s_{NQ} = select top samples in S_{vCPU}, S_{NQ} 14: 15: $diff_{vCPU} = TH'_{vCPU} - util_{vCPU} \text{ of } s_{vCPU}$ $diff_{NQ} = TH'_{NQ} - util_{NQ}$ of s_{NQ} 16: if $diff_{vCPU} \leq \tilde{d}iff_{NQ}$ then 17: 18: $TH'_{vCPU} = util_{vCPU}$ of s_{vCPU} 19: remove S_{vCPU} from S_{vCPU} and S_{NQ} 20: else $TH'_{NQ} = util_{NQ} \text{ of } s_{NQ}$ 21: 22: remove s_{NO} from S_{vCPU} and S_{NO} 23: end if 24: decCount = decCount - 125: end while $TH_{vCPU} = (1 - Weight) \times TH_{vCPU} + Weight \times TH'_{vCPU}$ 26: $TH_{NQ} = (1 - Weight) \times TH_{NQ} + Weight \times TH'_{NO}$ 27: 28: else if $count(S_{vio}) == 0$ then 29: $TH_{vCPU}, TH_{NO} = Reward \times TH_{vCPU}, Reward \times TH_{NO}$ 30: end if 31: Clear S 32: end if

 $util_{NO}$) (lines 12). The subsequent loop involves iterative adjustment of the thresholds for these violating samples (lines 13). For each top-ranked sample in the sorted sets S_{vCPU} and S_{NO} , it computes the differences between the temporary thresholds (TH'_{vCPU}) , TH'_{NO}) and the top-sample utilizations (lines 14-16). Importantly, it selects the sample with the smaller difference and updates the respective threshold, thereby removing this sample from both S_{vCPU} and S_{NO} (lines 17-23). This decision strategically avoids utilizations that have previously resulted in SLO violations. Additionally, it aims to change the threshold for either vCPUs or NOs in the direction with a smaller difference, thereby preventing underutilization due to excessively low thresholds. This iterative process continues until decCount is reduced to zero (lines 24-25). Subsequently, the final thresholds for vCPUs (Th_{vCPU}) and NQs (Th_{NO}) are calculated as a weighted average, factoring in both the new temporary thresholds and the previous values (lines 26-27). In situations without SLO violations, the thresholds are increased using the Reward factor (lines 28-30). Finally, the algorithm clears the set S, preparing for the next exploration round (line 31).

4.4 Implementation

Our implementation consists of vSPACE-Host and vSPACE-Guest operating within the hypervisor and the VM, respectively. vSPACE-Host within the hypervisor performs two primary tasks: dynamic core allocation for vCPUs and NQs and suitable thresholds exploration for vCPUs and NQs. It adjusts the number of active pNQs in order to match the number of pCPUs assigned to NQs. Conversely, the vSPACE-Guest within the VM is responsible for changing the number of active vCPUs and vNQs. To allocate pCPUs and modify the number of active vCPUs, we employ CPUSET, a feature of the Linux Cgroup system, which manages the range of available pCPUs and vCPUs. To change the number of NQs, we utilize RSS and XPS to manage the pNIC/vNIC, adjusting the number of pNQ/vNQ. The communication between these vSPACE-Host and vSPACE-Guest is facilitated by VSOCK, which supports zero-copy transfers. vSPACE periodically measures the utilization of pCPUs by NQs and vCPUs based on idle and non-idle time of the pCPUs.

5 Evaluation

5.1 Experimental Methodology

Our experimental setup involves testing with server and client systems connected via a 10 Gbps NIC, a configuration that ensures bandwidth is not a limiting factor for observing SLO violations. The server uses a twenty-core Intel Xeon Gold 6138 CPU and 192GB RAM, with the performance governor active and turbo boost off. The VM has 20 vCPUs and 16 GB memory, with the front-end driver and back-end device being the virtio-net driver and vhost-net, respectively [32, 41]. We use KVM hypervisor, with both the guest and host kernels operating on Linux 5.13.1.

Workloads We evaluate vSPACE with two LC workloads: memcached [24] and nginx [31]. memcached, an in-memory key-value store application, runs with 20 threads in the LC VM. The client machine generates network packets using mutilate, based on Facebook's ETC trace [4]. nginx is a web server application that consists of 100,000 1KB html files with 20 threads, and wrk on the client machine sends requests to read the *html* files on the server. We set 1ms and 10ms as the target SLO for memcached and nginx, respectively, established by the latency-load curve [16, 20]. To evaluate the resource efficiency of vSPACE-R, we run multi-threaded applications from parsec benchmark suite [6] on a VM as a BE workload (VM-BE): ferret, body track, freqmine, vips, fluidanimate. Additionally, we run SysBench with another BE workload, which conducts disk I/O read/write. In our co-allocation scenarios, we allocate the vCPUs of the VM-BE on idle pCPUs that are not allocated for vCPUs and NQs of the VM executing the LC workload (VM-LC). We run BE workloads only when evaluating the co-allocation scenarios for vSPACE-R and Demeter. The experimental methods remain consistent with those described in the motivation section. Comparative Studies. We evaluate the impact of vSPACE on performance, energy efficiency, and resource efficiency by comparison with three static configurations and three existing studies. We consider three static configurations: 20vCPU-1NQ and 20vCPU-20NQ for single or multiple NQs using a time-sharing approach, and SR-IOV, set for two NQs and 20 vCPUs. Note that the Intel 82599 ethernet controller, commonly used in Amazon AWS NICs for VM, has a two-queue limit per virtual function. In order to compare performance, we compare vSPACE with ES2, a state-of-the-art study in packet processing optimizations like adaptive hybrid vIRQ handling and vIRQ redirection, aiming to reduce VM exits [15]. In our comparison with ES2, we set the number of pNQs, vNQs, and vC-PUs to 1, 1, and 20, respectively, reflecting ES2's dependency on single packet processing. For the evaluation of energy efficiency,



we compare vSPACE with CoreNap, a state-of-the-art dynamic core allocation [28], which dynamically adjusts pCPUs for LC workload and packet processing to minimize active cores while satisfying the SLO. In CoreNap, threads of LC applications and packet processing for pNQs share pCPUs in a non-virtualization environment. To utilize CoreNap in virtualized environments, we slightly modify CoreNap. The modified CoreNap dynamically changes the number of pCPUs for vCPUs running LC workloads and pNQs/vNQs for packet processing, while the core allocation policy is the same as the original. We evaluate the resource efficiency of vSPACE in coallocation scenarios, wherein a VM running BE workloads (VM-BE) is allocated on idle pCPUs that are not allocated for VM executing LC workloads (VM-LC). This is contrasted with Demeter, which manages pCPUs and vCPUs for both VM types based on vCPU scheduling time for resource efficiency. Although Demeter also considers dynamic pCPU voltage and frequency management for VM-BE, our focus remains on resource efficiency, setting all pCPUs to their highest voltage and frequency states. For Demeter, we conduct experiments under two scenarios: single packet processing with 1NQ and parallel packet processing with 20NQs.

vSPACE Configuration. In vSPACE, we empirically set several parameters. To maintain stable core allocation, we've set the *MARGIN* to 1.1. The *Weight*, used for calculating subsequent thresholds as a weighted sum of previous ones, is set at 0.7. For The threshold exploration algorithm, *Reward* is used to increment the threshold when no SLO violations are observed among the collected samples; we set *Reward* to 1.03. vSPACE measures utilization and tail latency, performing dynamic core allocation every 100 ms. Threshold exploration begins after collecting 1000 samples (i.e., 100 seconds). Once stable thresholds are obtained through threshold exploration, we halt the online threshold exploration to achieve consistent results.

5.2 Comparison with static loads

Performance Analysis. Figure 8 depicts the maximum throughput representing the highest QPS under the SLO constraint in memcached and nginx. In this comparison, vSPACE-P shows the highest throughput, with an improvement of up to 318% in memcached and 33% in nginx compared to the 20vCPU-20NQ configuration. Meanwhile, vSPACE-E also shows considerable improvements similar to vSPACE-P. SR-IOV shows slight performance degration than vSPACE-P and vSPACE-E in memcached due to its parallism level limits (two-NQ). With ES2, performance enhancements are limited in memcached and even cause performance degradation in nginx.



This is because polling for packet processing affects scheduling VM running LC workloads; we will explain the details in the next paragraph. CoreNap's performance slightly improves as it reduces scheduling overhead by decreasing the number of vCPUs and NOs, even though it relies on a time-sharing approach.

Figure 9 and Figure 10 show system events across different dynamic core management policies in memcached and nginx at 50KQPS, where all dynamic core management policies satisfy the SLO. The system events include the number of pIRQs, vIRQs, context switches, VM exits, and migrations. vSPACE-P shows a reduction in the number of pIRQs/vIRQs compared to 20vCPU-20NQ at 50KQPS, showing decreases of 12.4%/23.3% in memcached and 12.9%/17.6% in nginx, respectively. This reduction of pIRQs and vIRQs moderates scheduling overhead, leading to a further decrease in the number of context switches and migrations by 34.5% and 98.9% in memcached at 50KQPS, respectively. vSPACE-E demonstrates even greater reductions in the number of pIRQs/vIRQs at 50KQPS in both memcached and nginx, with decreases of up to 40.7%/45.9% and 42.9%/49.0%, respectively. This reduction further decreases context switches, VM exits, and migrations by 75.2%, 59.4%, and 99.9% in memcached at 50KQPS, respectively. However, despite moderating the scheduling overhead, vSPACE-E shows higher P95 since a smaller number of pCPUs are utilized for vCPUs and NQs compared to vSPACE-P. SR-IOV offers reduced overhead (e.g., nearly zero pIRQs) by allowing VMs direct pNIC access, but performance is degraded by limited NQs. Although ES2 shows fewer system events compared to vSPACE-P, the performance remains limited or degrades since ES2 relies on packet processing with a single core, leading to pCPU utilization saturation; we delve into more details later. Furthermore, the polling method used in ES2 not only results in high CPU utilization but also misses the opportunity to batch packets due to rapid transmission processing; thereby, it is

physical IRQs (M) 0.0 0.0 0.0

Physical IRQs (M)

20vCPU-1NQ

20vCPU-20NQ

20vCPU-INQ

20vCPU-20NQ



10

Figure 11: Utilization of CPU-0, dedicated to pNQ packet processing and available for vCPUs or vNQs.

likely to impact VM running LC workloads negatively. Accordingly, ES2 increases the migration by up to 9.9×, showing performance degradation in nginx, compared to 20vCPU-1NQ. On the other hand, CoreNap reduces the number of pIRQs and vIRQs by reducing the number of pNQs and vNQs compared to the 20vCPU-20NQ. However, due to its time-sharing approach, CoreNap still incurs considerable scheduling overheads under high load.

In Figure 11, we observe the utilization of CPU-0, dedicated to pNO packet processing and available for vCPUs or vNOs. Notably, at 250 KQPS for memcached and 150 KQPS for nginx, ES2 approaches 95.3% and 95.4% utilization, respectively. Such saturation causes significant packet processing delays, leading to SLO violations. Moreover, at 250 KQPS for memcached, ES2 records 3.0× longer the total idle utilization compared to vSPACE-P, revealing its poor parallelism in packet processing. SR-IOV shows lower CPU-0 utilization due to its effective reduction of resource contention by bypassing typical virtualization pathways, although its hardware dependency raises concerns about availability and scalability. vSPACE-P shows unsaturated utilization, reserving CPU-0 only for



vSPACE-R



network packet processing. vSPACE-E shows a higher utilization than vSPACE-P at 50 KQPS due to fewer cores allocated for NQs compared to vSPACE-P.

Energy Efficiency Analysis. Figure 12 depicts the normalized energy consumption in memcached and nginx at specific load levels. We normalize all energy consumption values with respect to vSPACE-P, which exhibits the best performance at all load levels while excluding SLO violation cases. Our results show that vSPACE-E demonstrates the best energy reduction among all dynamic core management policies while satisfying the SLO for all load levels. At 50KQPS, where 20vCPU-20NQ satisfies the SLO, vSPACE-E reduces energy consumption by up to 19.6% and 24.4% compared to vSPACE-P and 20vCPU-20NQ respectively. SR-IOV significantly reduces energy consumption by decreasing utilization and contention, with the benefit from small NQs. CoreNap follows a timesharing approach, resulting in performance degradation caused by scheduling overheads in a virtualized environment. The performance degradation requires further active pCPUs to satisfy the SLO than vSPACE-E. On the other hand, vSPACE-E separately allocates pCPUs for vCPUs and NQs, effectively reducing the scheduling overhead and allowing for reserving additional idle pCPUs. For instance, vSPACE-E utilizes an average of 5 pCPUs in memcached at 50 KQPS while CoreNap activates an average of 9 pCPUs. Therefore, vSPACE-E shows energy reduction up to 16.5% compared to CoreNap. ES2 and 20vCPU-1NQ show reduced energy consumption at 50KQPS since they do not distribute the utilization for packet processing among pCPUs. However, the static number of vCPUs causes all pCPUs to be activated, preventing them from remaining idle. Lastly, 20vCPU-20NQ shows the highest energy consumption at 50KQPS, owing to maximizing the number of vCPUs and NQs. Resource Efficiency Analysis. As shown in Figure 13, the comparison of pCPU allocation under vSPACE-R and Demeter reveals significant differences in their approach to co-allocating resources

for VM-LC and VM-BE. We evaluate two types of Demeter configurations: one with a single NQ (i.e., Demeter(1NQ)) and one with multiple NQs (i.e., Demeter (20NQ)). vSPACE-R outperforms Demeter (1NQ) by allocating an average of 20.9% more pCPUs for memcached and 24.0% more for nginx, across 6 BE workloads at 50 KQPS. This advantage increases in the ferret benchmark, with vSPACE-R allocating 20% and 30.5% more pCPUs, respectively. Furthermore, vSPACE-R satisfies the SLO at 250KQPS in memcached and at 150KQPS in nginx, respectively. Conversely, Demeter satisfies the SLO only at 50KQPS; the allocated pCPUs for VM-BE are smaller than vSPACE-R due to overlooking packet processing.

Demeter(20NQ)

Demeter(20NQ)

Demeter(20NQ)

Demeter(20NQ)

(a) memcached at 50KQPS

(b) memcached at 150KQPS

(c) nginx at 50KQPS

SLO Vio

(d) nginx at 150KQPS

Demeter(1NQ)

Demeter(1NQ)

SLO Vio

Demeter(1NO)

Demeter(1NO)

0 Vio SLO Vio

Vumber of Cores

Number of Cores

Number of Cores

Number of Cores

LO Vio

5.3 **Comparison with Dynamic Loads**

To evaluate the dynamic core management capabilities of vSPACE-P and vSPACE-E under varying loads, we conduct experiments with dynamic load scenarios. We gradually increase the QPS from 50KQPS to 250KQPS, then subsequently decrease the QPS back to the initial QPS of 50KQPS. Figure 14 presents the average number of pC-PUs allocated for vCPUs and NQs as the load varies for vSPACE-P, vSPACE-E, and 20vCPU-20NQ every 20 seconds. The figure also shows normalized energy consumption and P95, measured every 20 seconds. The energy consumption is normalized to the maximum energy consumption that the processor can use, determined by its Thermal Design Power (TDP). vSPACE-P consistently satisfies SLO and exhibits low tail latency in most cases. vSPACE-P offers up to 64% lower latency at low loads compared to vSPACE-E, making it more suitable for applications requiring low latency. In vSPACE-P, the ratio of pCPUs allocated for vCPUs and NQs is slightly changed.



Figure 14: P95, Energy consumption and core allocation changes in memcached with a dynamic load.

Conversely, vSPACE-E adjusts the number of pCPUs for vCPUs and NQs proportionally as the QPS changes. vSPACE-E not only satisfies the SLO in all cases but also reduces energy consumption by up to 13.6% and 11.1% compared to 20vCPU-20NQ and vSPACE-P, respectively. Despite 20vCPU-20NQ utilizing all pCPUs at all times, SLO violations occur when the load surpasses 160 KQPS.

6 Related Work

Scheduling techniques. Several studies propose scheduling techniques to improve performance in I/O virtualization by reducing the scheduling overhead. vTurbo [37] assign vCPUs handling vIRQs to turbo cores with short scheduling intervals (e.g., 0.1 ms), significantly improving responsiveness by reducing hypervisor-induced scheduling delays. Similarly, Ahn et al. [1] prioritizes vCPUs in critical sections by allocating them to dedicated pCPUs, dynamically adjusting dedicated core numbers based on demand. CoINT [39] extends this concept, allowing critical-section vCPUs prolonged pCPU occupation. Meanwhile, eCS [17] employs para-virtualized signals for hypervisor notification, ensuring priority scheduling for critical tasks. Despite their potential, these approaches necessitate system modifications and largely ignore the performance bottlenecks introduced by parallel packet processing, limiting their applicability and overall performance enhancement in cloud environments. I/O handling techniques. Several studies propose I/O handling techniques to reduce I/O virtualization overhead. vBalance [10]

and hBalance [8] redirect vIRQs from inactive vCPUs to active ones, ensuring vIRQ handlers are promptly addressed without waiting for idle vCPUs to be rescheduled. ES2 [15] employs a hybrid I/O management method, switching between exit-based and polling modes to reduce VM exits, thereby mitigating exit-induced performance degradation. Nevertheless, these approaches ignore the advantages of employing parallel packet processing in vNIC. Such oversight can lead to underutilization of available resources and missed opportunities for further performance improvements in virtualized environments.

Dynamic resource management. Various studies focus on dynamic core allocation to improve resource efficiency without violating the SLO for LC workloads. Arachne[30] dynamically adjusts active threads and pCPUs based on utilization, allocating BE workloads to idle pCPUs for improved efficiency. vScale[9] manages vCPU distribution across VMs based on CPU demand, reducing scheduling delays and preventing performance degradation in resource-overcommitted environments. Advancements in core management policies, including dynamic voltage and frequency scaling (DVFS), balance both performance and power consumption. Hipster[26] and Twig[27] use reinforcement learning for dynamic core allocation and DVFS management for co-located workloads. Demeter [34] categorizes VMs to adjust core allocations and frequency scaling, prioritizing LC VM to manage energy consumption efficiently. Other studies consider multiple resources. Heracles[21], PARTIES[7], and Rhythm[40] manage CPU, memory, network, and disk resources, using feedback mechanisms for balanced utilization across LC and BE workloads. CLITE [29] adopts Bayesian optimization to predict near-optimal resource configurations, enhancing co-located workload performance. However, these studies often overlook parallel packet processing, leading to scheduling overheads and resource inefficiency.

Several studies focus on energy-efficient pCPU allocation. CARB[38] adjusts pCPUs based on the slack between the SLO and response latency to minimize active pCPUs for LC workloads. Peafow1[3] uses a heuristic based on request rates to preserve idle pCPUs while meeting the SLO. CoreNap [28] addresses this by adjusting pCPU allocations for both LC workloads and packet processing, informed by latency and consumption forecasts. However, these neglect the complexities of I/O virtualization, resulting in energy inefficiency. Hardware Acceleration Approach. SR-IOV [12] provides direct VM access to physical I/O devices, reducing virtualization overhead. Similarly, Microsoft Azure's VMMQ [25] combines SR-IOV with RSS to distribute network traffic across multiple cores. These hardware solutions offer performance benefits but also present challenges, such as during VM migration due to reliance on specific hardware capabilities. Our approach, vSPACE, contrasts with hardware-dependent methods by offering a flexible, software-based solution. Unlike static NQ management in hardware solutions, vSPACE dynamically manages NQs, avoiding inefficiencies from over-allocating NQs and unnecessary pCPU activation. Additionally, hardware constraints like the Intel 82599 ethernet controllers' two-queue limit per virtual function [5], commonly used in Amazon AWS NICs, highlight vSPACE's advantage. Its software-driven approach bypasses such limitations, ensuring efficient vCPU resource utilization without hardware-specific restrictions. SmartNICs [22], using a hardware-centric approach, benefit by offloading network

tasks such as packet filtering, routing, and IP translation. However, some tasks, such as memory copying between the kernel/hypervisor and VMs, cannot be offloaded. vSPACE aims to mitigate these inefficiencies through a software-centric approach. Therefore, combining SmartNICs and vSPACE offers an opportunity to improve performance and efficiency further. We will explore this in future work.

7 Conclusion

Datacenters face significant challenges in parallel processing within virtualized environments, particularly risking SLO violations for LC workloads. Previous studies mitigate I/O overheads but often overlook the impact of excessive parallelism, which leads to complex scheduling challenges, such as frequent preemptions and migrations due to high pCPU demand. To mitigate these challenges, we introduce vSPACE, dynamic core management designed to support parallel network packet processing in virtualized environments. vSPACE employs two main strategies: it ensures independent pCPU allocation for vCPUs and NQs, and it dynamically adjusts this allocation based on periodic pCPU utilization monitoring. This allows vSPACE to proactively identify situations where pCPU utilization is at risk of saturation, ensuring an appropriate number of pC-PUs are allocated to both vCPUs and NQs as needed. Moreover, vSPACE offers three operational modes to enhance performance (vSPACE-P), energy efficiency (vSPACE-E), and resource efficiency (vSPACE-R), catering to the diverse needs of data centers. Our evaluations show that vSPACE significantly improves throughput (i.e., maximum QPS) by up to 4.2× compared to previous policies for core allocation. Additionally, vSPACE offers considerable improvements in energy efficiency and resource utilization by up to 16.5% and 30.5% compared to state-of-the-art dynamic core allocation techniques.

Acknowledgments

This research was supported in part by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00503, Researches on next generation memory-centric computing system architecture), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2022-0-00498, Development of high-efficiency AI computing SW core technology for high-speed processing of large learning models), Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00396013, DRAM PIM Hardware Architecture for LLM Inference Processing with Efficient Memory Management and Parallelization Techniques), and National Research Foundation of Korea (NRF) grants funded by the Korean government (MSIT) under Grant (No. 2018R1A5A1060031).

References

- Jeongseob Ahn, Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. 2018. Accelerating critical OS services in virtualized systems with flexible micro-sliced cores. In European Conference on Computer Systems (EuroSys). 1–14.
- [2] Amazon. [n. d.]. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/ instance-types/#instance-details.
- [3] Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, and Timothy Zhu. 2020. Peafowl: in-application CPU scheduling to reduce power consumption of inmemory key-value stores. In ACM Symposium on Cloud Computing (SoCC). 150– 164.

- [4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In ACM SIGMET-RICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems. 53–64.
- [5] AWS. 2024. Enable enhanced networking with the Intel 82599 VF interface on Linux instances. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/ latest/UserGuide/sriov-networking.html.
- [6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques. 72–81.
- [7] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 107–120.
- [8] Luwei Cheng and Francis CM Lau. 2016. Offloading interrupt load balancing from smp virtual machines to the hypervisor. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 27, 11 (2016), 3298–3310.
- [9] Luwei Cheng, Jia Rao, and Francis CM Lau. 2016. vscale: Automatic and efficient processor scaling for smp virtual machines. In ACM European Conference on Computer Systems (EuroSys). 1–14.
- [10] Luwei Cheng and Cho-Li Wang. 2012. vBalance: using interrupt load balance to improve I/O performance for SMP virtual machines. In ACM Symposium on Cloud Computing (SoCC). 1–14.
- [11] Yaozu Dong, Dongxiao Xu, Yang Zhang, and Guangdeng Liao. 2011. Optimizing network I/O virtualization with efficient interrupt coalescing and virtual receive side scaling. In *IEEE International Conference on Cluster Computing (CLUSTER)*. 26–34.
- [12] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. J. Parallel and Distrib. Comput. 72, 11 (2012), 1471–1480.
- [13] Google. [n. d.]. Google Cloud Virtual Machine Types. https://cloud.google.com/ compute/docs/machine-types.
- [14] HaiBing Guan, YaoZu Dong, RuHui Ma, Dongxiao Xu, Yang Zhang, and Jian Li. 2012. Performance enhancement for network I/O virtualization with efficient interrupt coalescing and virtual receive-side scaling. *IEEE Transactions on Parallel* and Distributed Systems (TPDS) 24, 6 (2012), 1118–1128.
- [15] XiaoKang Hu, Jian Li, Ruhui Ma, and Haibing Guan. 2020. ES2: Building an Efficient and Responsive Event Path for I/O Virtualization. *IEEE Transactions on Cloud Computing (TCC)* (2020).
- [16] 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 IEEE/ACM International Symposium on Microarchitecture (MICRO). 143–154.
- [17] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2018. Scaling Guest {OS} Critical Sections with {eCS}. In USENIX Annual Technical Conference (ATC). 159–172.
- [18] Jian Li, Ruhui Ma, HaiBing Guan, and David SL Wei. 2015. vINT: Hardwareassisted virtual interrupt remapping for SMP VM with scheduling awareness. In IEEE International Conference on Cloud Computing Technology and Science (CloudCom). 234–241.
- [19] Jian Li, Shuai Xue, Wang Zhang, Ruhui Ma, Zhengwei Qi, and Haibing Guan. 2017. When I/O interrupt becomes system bottleneck: Efficiency and scalability enhancement for SR-IOV network virtualization. *IEEE Transactions on Cloud Computing* 7, 4 (2017), 1183–1196.
- [20] 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.
- [21] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In ACM SIGARCH Computer Architecture News, Vol. 43. 450–462.
- [22] Mellanox. 2018. http://www.mellanox.com/page/products_dyn?product_family= 275&mtag=bluefield_smart_nic.
- [23] Mellanox Mellanox. 2020. Mellanox ConnectX-5 product brief. [Online]. Available: https://www.mellanox.com/related-docs/prod_adapter_cards/ PB_ConnectX-5_EN_Card.pdf.
- [24] Memcached. [n. d.]. https://memcached.org. Accessed on 04/30/2019. https: //memcached.org.
- [25] Microsoft. 2023. Overview of Virtual Machine Multiple Queues (VMMQ). [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/ network/overview-of-virtual-machine-multiple-queues.
- [26] 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.
- [27] 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.

- [28] Gyeongseo Park, Ki-Dong Kang, Minho Kim, and Daehoon Kim. 2022. CoreNap: Energy Efficient Core Allocation for Latency-Critical Workloads. *IEEE Computer Architecture Letters (CAL)* 22, 1 (2022), 1–4.
- [29] Tirthak Patel and Devesh Tiwari. 2020. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 193–206.
- [30] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: core-aware thread management. In USENIX Symposium on Operating Systems Design and Implementation (OSDI). 145-160.
- [31] Will Reese. 2008. Nginx: the high-performance web server and reverse proxy. Linux Journal 2008, 173 (2008), 2.
- [32] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. ACM SIGOPS Operating Systems Review 42, 5 (2008), 95–103.
- [33] Stijn Schildermans, Jianchen Shan, Kris Aerts, Jason Jackrel, and Xiaoning Ding. 2021. Virtualization overhead of multithreading in X86 state-of-the-art & remaining challenges. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 32, 10 (2021), 2557–2570.
- [34] Wenda Tang, Yutao Ke, Senbo Fu, Hongliang Jiang, Junjie Wu, Qian Peng, and Feng Gao. 2022. Demeter: QoS-aware CPU scheduling to reduce power consumption of multiple black-box workloads. In ACM Symposium on Cloud Computing (SoCC). 31–46.

- [35] Willem de Bruijn Tom Herbert. [n. d.]. Scaling in the Linux Networking Stack. https://static.lwn.net/kerneldoc/networking/scaling.html
- [36] Jason Wang. [n. d.]. Multiqueue virtio-net. https://www.linux-kvm.org/page/ Multiqueue
- [37] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. 2013. {vTurbo}: Accelerating Virtual Machine {I/O} Processing Using Designated {Turbo-Sliced} Core. In USENIX Annual Technical Conference (ATC). 243–254.
- [38] 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.
- [39] Wang Zhang, Xiaokang Hu, Jian Li, and Haibing Guan. 2018. CoINT: Proactive Coordinator for Avoiding Interruptability Holder Preemption Problem in VSMP Environment. In IEEE International Conference on Computer Communications (INFOCOM). 477–485.
- [40] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: component-distinguishable workload deployment in datacenters. In ACM European Conference on Computer Systems (EuroSys). 1–17.
- [41] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. ACM SIGCOMM Computer Communication Review 45, 4 (2015), 523–536.