Taiji: A DPU Memory Elasticity Solution for In-production Cloud Environments

Hao Zheng[1*], Longxiang Wang[2*], Yun Xu[1], Qiang Wang[2], Yibin Shen[1], Xiaoshe Dong[2], Bang Di[1], Jia Wei[2], Shenyu Dong[1], Xingjun Zhang[2], Weichen Chen[1], Zhao Han[3], Sanqian Zhao[1], Dongdong Huang[1], Jie Qi[1], Yifan Yang[1], Zhao Gao[1], Yi Wang[1], Jinhu Li[1], Xudong Ren[1], Min He[1], Hang Yang[1], Xiao Zheng[1], Haijiao Hao[1], Jiesheng Wu[1]

[1] Alibaba Cloud, Hangzhou,, China

[2] School of Computer Science, Xi'an Jiaotong University, Xi'an,, China

[3] Department of Computer Science, Shanghai Jiao Tong University, Shanghai,, China

Abstract

The growth of cloud computing drives data centers toward higher density and efficiency. Data processing units (DPUs) enhance server network and storage performance but face challenges such as long hardware upgrade cycles and limited resources. To address these, we propose Taiji, a resource-elasticity architecture for DPUs. Combining hybrid virtualization with parallel memory swapping, Taiji switches the DPU's operating system (OS) into a guest OS and inserts a lightweight virtualization layer, making nearly all DPU memory swappable. It achieves memory overcommitment for the switched guest OS via high-performance memory elasticity, fully transparent to upper-layer applications, and supports hot-switch and hot-upgrade to meet in-production cloud requirements. Experiments show that Taiji expands DPU memory resources by over 50%, maintains virtualization overhead around 5%, and ensures 90% of swap-ins complete within 10μs. Taiji delivers an efficient, reliable, low-overhead elasticity solution for DPUs and is deployed in large-scale production systems across more than 30,000 servers.

1 Introduction

To achieve low latency and high throughput in I/O processing, major cloud providers deploy DPUs to offload network, storage, and control tasks, allowing CPUs to focus on computation and improving system performance and efficiency [12, 60]. However, rapid growth in e-Commerce, mobile and AI applications drives an exponential demand for I/O, memory, and computing resources [55], increasing the requirements of DPU resources. Rising VM demands for peak IOPS and bandwidth, combined with higher VM density on cloud servers, strain DPU resource allocation. Current mainstream DPU models provide only 32GB of memory (e.g., BlueField-3 [39]), struggling to meet growing service demands. Furthermore, DPU hardware upgrades are impractical due to long cycles and high costs in deployed systems [37].

Our observations in the cloud show that compute nodes rarely fully use their maximum storage IOPS and network PPS capabilities, with simultaneous peak utilization rare. DPU resources reserved for peak demand are often underutilized [47–49]. Memory is the primary infrastructure cost in current data centers [58]. Our production data indicates that a page accessed in 10 minutes is hot, with more than 90% of DPUs having a cold page water level above 50%. Thus, resource elasticity is a viable approach to address the mismatch between DPU hardware and software resource demands. Memory elasticity can be achieved through swapping or virtualization, but existing solutions face challenges in DPU:

1) **Memory swapping functionality and efficiency need enhancement.** High-performance DPU services, such as storage and networking, heavily rely on HugeTLB huge pages. Current mainstream solutions mainly support small pages and have limited large-scale deployment [23, 46, 50, 51, 65]. Newer kernels support THP for huge page swapping, but splitting THP during swap-out impedes later huge page allocation [33, 40, 44]. Moreover, a significant portion of kernel memory remains unswappable [1].

2) **High resource overhead and adaptation cost of virtualization solutions.** Virtualization enables memory swapping in VMs but incurs substantial resource costs beyond performance impact [6, 17, 42, 54, 63]. Both Type 1 hypervisors requiring a privileged VM and Type 2 hypervisors relying on a Host OS must run an additional OS, which is costly under DPU resource constraints [30, 56]. Migrating existing DPU workloads into a virtualized environment further requires high adaptation effort and risks lengthening I/O paths, potentially degrading native performance [30].

Therefore, a lightweight software solution is needed to address DPU resource insufficiency. We present Taiji, a lightweight system for DPU resource elasticity. Taiji inserts a minimal virtualization layer between the host OS and hardware at runtime, enabling full hardware control while delivering efficient elastic resources to the OS. The design ensures no performance degradation, meets growing resource demands, and supports seamless deployment across large-scale, existing DPU servers. To our knowledge, this is the first paper to use virtualization for memory elasticity in DPUs. Our key contributions are:

Lightweight Hybrid Virtualization on DPUs. We propose a lightweight hybrid virtualization architecture for

[*] Hao Zheng and Longxiang Wang contributed equally to this work.

DPUs that achieves full memory swappability with minimal virtualization and resource overhead. By specialized address-space management, it switches the Host OS to a Guest OS and inserts a low-overhead virtualization layer without increasing resource overhead on the Host OS. It also minimizes resource overhead by sharing the user-space layer with the Guest OS, avoiding an extra user-space layer.

High Performance DPU Memory Elasticity. Targeting the widely used huge page in DPU environments, a high performance memory elasticity solution is designed at the huge page granularity, expanding virtual memory by more 50%. Furthermore, considering the characteristics of the IO path in DPUs, we dynamically filter the potentially involved memory in DMA and provide efficient memory elasticity through parallelized LRU and SWAP design, ensuring rapid swap-in, with 90% of page faults resolved in under 10 µs.

Seamless DPU resource elasticity scheduling. Leveraging the distinct roles of data-plane processors (DPs) and control-plane processors (CPs) on DPUs, Taiji uses the CPs to allocate compute resources to elasticity tasks via priority coordination and fair time-slice allocation. This schedules front-end CPU and back-end memory elasticity tasks while minimizing interference with DP high-performance I/O. The unified scheduler also reserves interfaces for extending CPU resource elasticity.

Large-scale Online DPU Deployment Practice. To ensure compatibility with existing running DPUs, Taiji uses a modular design that needs no changes to the running OS, allowing seamless hot-switch of running DPUs while maintaining normal service operation and enabling elastic resource capabilities. Its hot-upgrade feature also enables rapid deployment, updates, and evolution, enhancing the reliability and scalability of DPU resource elasticity. Taiji is currently running stably online, deployed at scale on over 30,000 DPUs.

2 Background and Motivation

2.1 Current Resource Usage of DPUs

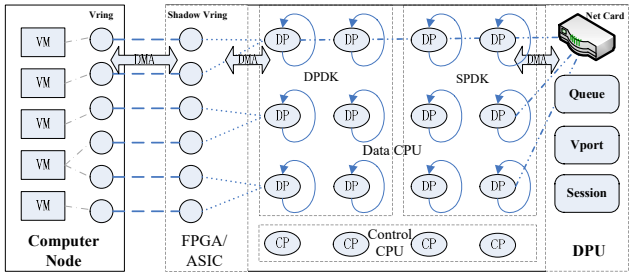


Figure 1. The DPU Resource Usage

2.1.1 DPU Usage Characteristics. In cloud computing, the `virtio` protocol [68] is widely used in the I/O path, particularly on DPUs [3, 9, 39]. As shown in Figure 1, the frontend `vring` on the compute node and the backend `shadow vring` on the DPU exchange data via DMA. The DPU's `vring` connects to corresponding polling DPs through DMA, with DPDK or SPDK interacting with the hardware NIC via DMA. Numerous DMA operations occur along the I/O path, and DPDK/SPDK continuously poll DPs for fast data processing, which is critical for high DPU performance. To support this, **DPU CPUs are primarily used for polling data-plane requests, with only a few reserved for control-plane processes,** and **the high-speed I/O path's multiple data transfers require extensive DMA operations and large amounts of memory involved in DMA.**

Moreover, the data plane of DPDK and SPDK extensively uses `hugetlb` and requires it to be as contiguous as possible to improve address translation efficiency. For network and storage services, substantial memory is also allocated to control-plane data, including network functions such as `vports` and `sessions` for access control, rate limiting, traffic statistics, and fast forwarding, and storage functions such as `queues` for device and queue management, I/O error recovery, second-level monitoring, and traffic statistics. For performance reasons, memory used by the data plane and control plane relies heavily on `hugetlb`. The amount of such memory depends heavily on the DPU's supported specifications—the higher the service specification, the greater the memory consumption. However, overall DPU resources are limited; for example, early hardware had only 16 GB of memory, and mainstream versions now have 32 GB. **As product specifications grow, DPU resource usage remains tight even with hardware upgrades.**

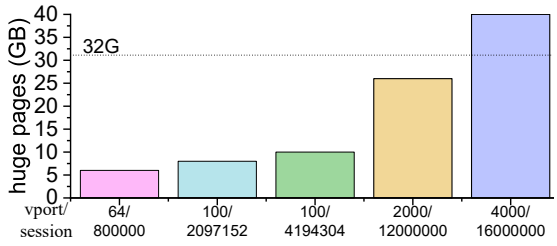
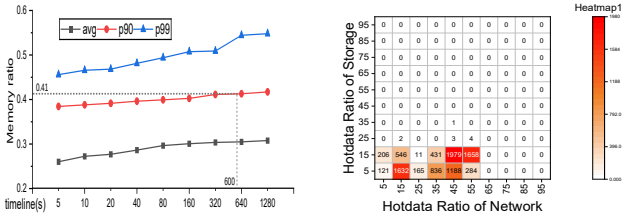


Figure 2. The DPDK Huge Page Usage

2.1.2 Static Memory Usage. While total memory consumption is complex and involves various data structures, memory usage per control data type in the same version can be approximated as `f(n) = base + each × n`, where `n` is the number of instances (e.g., `vport`, `session`, `queue`), `base` is the base memory to enable the function, and `each` is the per-instance overhead—e.g., each storage `queue` uses ~2.7 MB, and each network requires ~1 GB per 2 million sessions. This overhead may vary across versions due to added features or optimizations. As shown in Figure 2, DPDK reserves memory for service metrics, with consumption increasing as specifications grow. Resources are often preallocated to peak levels to handle workload spikes and avoid allocation failures from fragmentation, potentially causing waste. Moreover, during hot upgrades, `dpdk` and `spdk` require data migration

between old and new processes, necessitating an additional equal amount of memory reserved in advance (currently 8 GB on a 32 GB DPU). Thus, **to support maximum product specifications, storage and network services preallocate large amounts of statically allocated memory, with an additional equal amount reserved for hot upgrades.**

2.1.3 Dynamic Memory Usage. As shown in Figure 3a, although much memory is statically allocated in huge pages to DPDK and SPDK for peak performance, most remains idle: the 10-minute average cold page ratio exceeds 70%, and P90 exceeds 50%. As shown in Figure 3b, DPDK and SPDK rarely peak simultaneously. Based on one-hour data from over 7,000 servers, both are rarely busy—network activity is mostly below 60%, storage below 20%—with no observed simultaneous peaks. **Under real workload pressure, the utilization of statically allocated memory is low, with many cold pages, and storage and network services rarely reach peak performance simultaneously, with only occasional single-side high load.**



(a) The DPU Hot Data Ratio **(b)** The DPU Hot Data Hotmap

Figure 3. Comparison of DPU hot data metrics.

***Finding 1**: A large amount of huge page on the DPU, reserved for peak performance, remains unused in practice.*

2.2 Challenges in Existing Solutions

2.2.1 Functionality and Efficiency Challenges with Memory Swapping. Memory usage patterns in DPUs are closely tied to running services, making memory utilization changes difficult. For example, storage and network services often use huge pages like HugeTLB for high performance, with our online data frequently exceeding 75%. Traditional memory swapping is rarely implemented at scale and mainly targets small pages [65], making huge page swapping impractical. While swapping techniques have evolved to support THP [14], the predominant use of HugeTLB on DPUs—which remains unswappable—results in over 75% of memory being non-swappable. Using traditional swapping methods limits expandable memory resources, failing to meet the expansion needs of business software on the DPU. Even if the OS is modified to support HugeTLB swapping, many kernel memory areas, such as slab and crashkernel, remain unswappable, preventing full use of cold memory.

Significant latency affects current memory swapping techniques. Even with memory backend, zswap [25, 58] incurs 40µs average latency for 4KB pages. Swapping 2MB huge pages typically takes milliseconds [16, 57]. Failed huge page allocation triggers prolonged reclamation, further increasing latency. Fallback to small pages during swap severely degrades performance. Thus, existing solutions fail to meet DPU requirements [31], especially for storage and network services requiring 90% swap-in latency under 10µs. **A high-performance memory swapping solution is essential to preserve huge page benefits while meeting strict latency targets.**

Table 1. Comparison of Memory Elasticity Solutions

Memory Elasticity Solution		Swappable Memory	Unswappable Memory	Resource Overhead	Open Source	Swapping Mechanism
Memory Swapping	Traditional swap	User-space small pages	Various huge pages, kernel pages	Low	Yes	zram, ksm, swap
	THP swap	User-space small pages, THP	HugeTLB, kernel pages	Low	Yes	THP huge pages
VM-Based Solutions	KVM	Full	None	High	Yes	**Host-level**: Rely on host OS or hypervisor. **Guest-level**: Rely on guest OS. **PV**: Balloon.
	Xen	Full	None	High	Yes	
	Hyper-V	Full	None	High	No	
	VMware ESXi	Full	None	Low	No	

2.2.2 Lightweight and Adaptability Challenges with Virtualization. Virtualization enables resource elasticity [42] via full VM memory swapping. However, mainstream Type 2 KVM [63] requires a Host OS, incurring fixed overhead on a 32GB DPU—at least 524MB for struct page, 256MB for crash kernel, and over 600MB for the OS—about 5% of memory, excluding virtualization overhead. Running two OSes doubles stability and security risks [62]. Type 1 systems like Xen [6] and Hyper-V [54] require a privileged VM, incurring similar costs. VMware ESXi [17] avoids this but is closed-source and not directly usable. These approaches rely on Host OS or hypervisor swapping for full Guest OS memory swap, facing the same performance issues.

Moreover, DPU services rely on stable DMA operations and direct hardware interaction, making seamless migration to VMs difficult. Adapting them for VM execution requires significant modifications, leading to high adaptation costs and potential stability issues. Running these services in VMs may lengthen I/O paths, degrade performance, and require extensive optimization. Operating two OSes also increases maintenance overhead. Thus, maintaining dual OS instances raises operational complexity, stability risks, and security vulnerabilities. **Deploying traditional virtualization on resource-constrained DPUs incurs prohibitive costs, necessitating a lightweight virtualization approach tailored for DPU resource elasticity.**

***Finding 2**: Current memory elasticity solutions cannot be directly applied to DPU environments due to limitations in functionality, performance, or resource overhead.*

2.3 Challenges in Production Environments

2.3.1 Issues with Existing Online Deployments. Our public cloud has deployed hundreds of thousands of DPUs, supporting many businesses that need resource elasticity to take advantage of the latest software updates and higher product specifications. Traditional memory swapping and virtualization solutions [24] cannot ensure transparency for online businesses and often require a DPU reboot to enable resource elasticity, disrupting user service continuity. While live migration can reduce impacts on businesses, the cost and duration of large-scale rolling upgrades are high, and it still does not offer a seamless experience for users [11], especially those with high-load and sensitive requirements, leading to potential service jitter.

Another key scenario for DPUs is providing secondary virtualization for bare metal servers [67]. Currently, there are no effective solutions for live migration of bare metal instances, which represent over 30% of online deployments and often demand higher performance. Additionally, public clouds have strict SLO guarantees for stable and reliable services, making large-scale reboots impractical for implementing resource elasticity. Thus, solutions requiring DPU reboots cannot be deployed on existing servers. **A seamless hot-deployment resource elasticity solution is needed to fully leverage resource elasticity benefits.**

2.3.2 Maintainability and Extensibility of Elasticity Solutions. In cloud computing, every component requires upgrades and maintenance, making hot upgrade capability essential [66]. Without it, even a resource elasticity component that supports seamless deployment for existing and incremental DPUs will face challenges during future upgrades that affect users. Hot upgrades help resolve faults exposed by large-scale operations and allow for easy extension of new features (like CPU elasticity), enabling rapid release and iteration of the resource elasticity component in small, quick steps. Thus, **our resource elasticity architecture must include hot upgrade capabilities to quickly realize and expand the benefits of resource elasticity.**

Moreover, using separate solutions for incremental and existing DPU deployments increases maintenance costs and risks to online stability. Meanwhile, to ensure high performance, DPUs allocate most CPU resources to data-plane tasks, leaving limited capacity for control-plane tasks, which may become constrained as service scales grow. Traditional CPU elasticity methods often require OS scheduler modifications [20, 34, 53], leading to maintenance challenges similar to those in memory elasticity. To support future upgrades and maintenance, **we seek a unified virtualization-based resource elasticity solution that minimizes components and operational complexity**, serving both new and existing DPUs while addressing immediate memory elasticity and enabling future CPU elasticity.

> ***Finding 3**: Fully realizing the benefits of resource elasticity requires hot-switch and hot-upgrade capabilities for large-scale deployment on both existing and new DPUs.*

3 Overview

3.1 Objectives

Based on the above analysis, DPUs statically reserve substantial resources for peak demands that often remain idle, presenting an opportunity for elasticity. Meanwhile, given limitations of existing solutions and production maintenance challenges, elasticity must meet the following requirements:

O1 - Lightweight: DPU resources are extremely constrained, so the elasticity solution's overhead must be minimal—around 1% of DPU resources—and must not degrade the DPU's native performance, especially ensuring less than 5% peak performance loss for storage and network services.

O2 - High Efficiency: The solution must be highly efficient with minimal impact on existing services; in particular, to avoid degrading high-performance I/O, the 90th percentile page fault latency must be under 10µs; additionally, due to extensive DMA usage, memory elasticity must not introduce data correctness risks.

O3 - High Benefit: Given the DPU's limited resources, the elasticity solution must deliver substantial resource gains—we target over 50% memory elasticity—while supporting both routine operations and sudden high-load workloads.

O4 - Transparency: To accommodate large-scale online DPUs, the solution must enable transparent deployment without disrupting existing services, and support transparent upgrades for seamless large-scale rollout.

3.2 Architecture

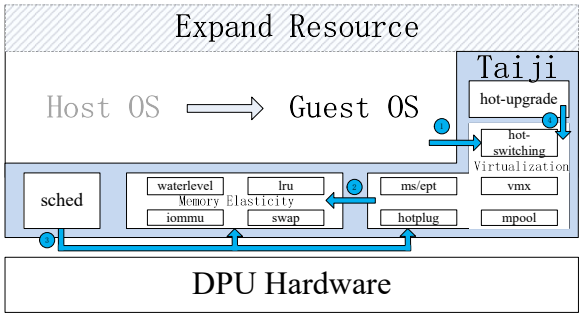


Figure 4. The Architecture of Taiji.

As shown in Figure 4, Taiji consists of four main modules: the Virtualization Layer enables basic virtualization functions, Memory Elasticity provides resource expansion, the Scheduler coordinates Guest OS and elastic task execution, and Hot Upgrade supports online upgrades. The main functionality of each module are as follows:

3.2.1 Virtualization Layer. Any resource elasticity solution must not worsen constraints, especially on memory-limited DPUs. DPU virtualization here does not manage multiple VMs—it only virtualizes the existing OS to overcome physical limits. To avoid OS overhead, Taiji hot-switches the DPU's OS online (**O4**) to insert a minimal virtualization layer, achieving lightweight virtualization (**O1**) and enabling resource elasticity (1). After switching, the Host OS becomes a Guest OS, and Taiji controls the hardware. From the virtualization layer's view, all Guest OS memory becomes swappable, enabling memory elasticity. Virtual memory is then hot-plugged into the Guest OS, allowing time-division multiplexing of physical memory (**O3**).

3.2.2 Memory Elasticity. Due to high-performance DPU service requirements, memory elasticity must be efficient (**O2**) without performance loss. Taiji tracks physical memory hot/cold status using a multi-level LRU and multi-task concurrency to speed data collection. It proactively swaps cold pages to backend storage via parallel SWAP to enable Guest OS memory elasticity (2). A concurrent swap-in design keeps P90 swap-in latency under 10µs. As much DPU memory may be involved in DMA, the swapping process avoids DMA memory. A watermark-based policy controls swapping to prevent thrashing, reserve free memory for sudden spikes, and ensure stable operation (**O3**).

3.2.3 Resource Scheduling. With memory elasticity enabled, Taiji runs both switched front-end VCPU and background elasticity tasks. To prevent background tasks from affecting front-end services, especially on DPs, a scheduler coordinates execution (3). Based on DPU CPU usage, CPs let original services yield time slices to elasticity tasks. Multi-task memory elasticity also requires parallel scheduling to meet **O2**. The scheduler coordinates execution using static configuration and dynamic adjustment: assigning priorities, allocating time slices, and adjusting them based on runtime feedback to ensure fair scheduling across priorities.

3.2.4 Hot Upgrade. To maximize DPU resource elasticity, Taiji is designed to support large-scale existing online DPUs from the outset. Its virtualization layer enables hot-switch for seamless online transition to the elastic architecture. As a core DPU component, Taiji also supports hot-upgrade (4) through a modular, compatibility-driven design for future updates and bug fixes. With hot-switch and hot-upgrade (**O4**), Taiji can run on all existing and new DPUs as a unified resource management platform without disrupting running services, enabling large-scale production rollout.

4 Design and Implement

4.1 Virtualization Layer

4.1.1 Lightweight Hybrid Virtualizationl. To achieve **O1**—a low-overhead lightweight virtualization solution—Taiji designs the virtualization layer to reduce both resource and performance overhead. **Performance overhead** is minimized using register passthrough and huge page support. Taiji's Virtual Machine Control Structure (VMCS) design applies register passthrough whenever possible to avoid costly VM exits, removing most virtualization overhead [26, 32]. As huge pages are common in DPUs, the virtualization layer's memory management retains this feature to avoid address translation overhead, with the underlying extended page table (EPT) using huge pages to reduce TLB miss impact [2, 8, 22]. **Resource overhead** reduction requires a new, more complex hybrid virtualization architecture. As shown in Figure 5, Taiji switches physical CPUs (PCPUs) to virtual CPUs (VCPUs), hot-switch the original Host OS into a Guest OS and inserting a thin virtualization layer. The switched Host OS runs in non-root mode as the Guest OS, while the Taiji module resides in the Guest OS but operates mainly in root mode to manage hardware (Host layer), with a small interface layer in non-root mode (Guest layer). This design takes over hardware resource management while reusing the Guest OS user space, avoiding the need for an additional user layer, and forms a hybrid virtualization architecture distinct from traditional Type 1 and Type 2 designs. Specifically, it has the following particularities:

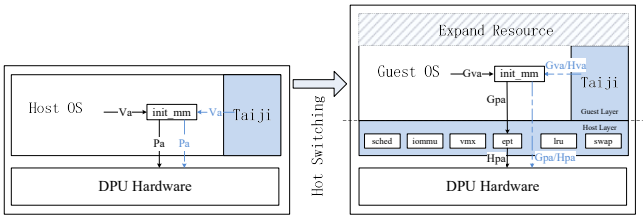


Figure 5. The Hybrid Virtualization Architecture

First, Taiji is implemented as a module inserted into the OS to run both before and after the virtualization switch. Before the switch, Taiji functions as a regular kernel module. After the switch, the Host OS becomes the Guest OS, but Taiji remains a module in the Guest OS while taking over the underlying hardware. Most of Taiji runs in root mode, with a small interface part in non-root mode, and both share the Guest OS's user space.

Second, since the Taiji module runs inside the OS kernel, they share the same kernel address space and thus the same virtual addresses. Before the switch, both use the OS virtual address (VA). After the switch, the OS uses the Guest Virtual Address (GVA) in non-root mode, and Taiji uses the Host Virtual Address (HVA) in root mode. Sharing the same kernel address space results in GVA = HVA.

Third, because Taiji runs inside the OS, its address translation also uses the kernel page table (`init_mm`) to convert VA to PA. Before the switch, this is identical to the Host OS. After the switch, memory access paths diverge: when the Guest OS accesses memory, its GVA is first translated via `init_mm` into a Guest Physical Address(GPA), which is then

converted by EPT into an Host Virtual Address (HPA) to access physical memory.

Fourth, after the switch, when the Taiji module accesses physical memory, it still uses `init_mm` to translate its VA into a GPA. The interface part running in the Guest passes through EPT to obtain the HPA. However, since most of the Taiji module runs in root mode, the main virtualization layer bypasses EPT and directly accesses physical memory via GPA. This single-layer page table translation improves memory access performance [64] but requires GPA = HPA to ensure correctness.

The most distinctive feature of this virtualization is its address space management. Running inside the Guest OS while acting as a hypervisor, the Taiji module and Guest OS have **GVA = HVA**, fundamentally different from traditional architectures. Since Taiji accesses physical memory through a single-layer page table, correctness requires all **Taiji** accessed memory be **GPA = HPA**. Based on this, we design a metadata pool (`mpool`) that allocates full pages and slab memory at various granularities. All Taiji metadata is allocated from this pool, whose memory is pinned and excluded from swapping, ensuring GPA = HPA and forming the basis of the hybrid virtualization architecture. Centralized metadata management also prevents fragmentation, avoiding negative effects on future memory elasticity. Other Taiji functions are built on this pool.

4.1.2 Virtualization Switching.

To achieve **O4**, i.e., deploying Taiji on running DPUs, the Host OS is hot-switched into a Guest OS by converting each PCPU to a VCPU. For every running PCPU, an EPT table, a VMCS, and other virtualization structures are prepared to minimize overhead. The switch is triggered by an `SMP` call to enter virtualization mode. Upon receiving the call, the target CPU executes `switch_vcpu` in two stages. First, the PCPU saves registers, switches to the VCPU stack, transfers the saved state into the VMCS (1), and calls `hv_sched`, which invokes `vcpu_run`. This loads the VMCS and registers, then executes `VMLAUNCH` to enter non-root mode. The VCPU's first instruction re-invokes `switch_vcpu`, which runs the second stage, restores the stack, and completes the switch (2). The VCPU then resumes from the PCPU's execution flow, returning from the `SMP` call. The Guest OS CPU continues as a VCPU until a privileged instruction or exit event triggers a VM exit (3), while the PCPU in root mode runs `hv_sched` to schedule VCPUs and elastic memory tasks (Section 4.3, 4). This process repeats for all PCPUs until the switch is complete, enabling a hot-switch to the virtualization architecture.

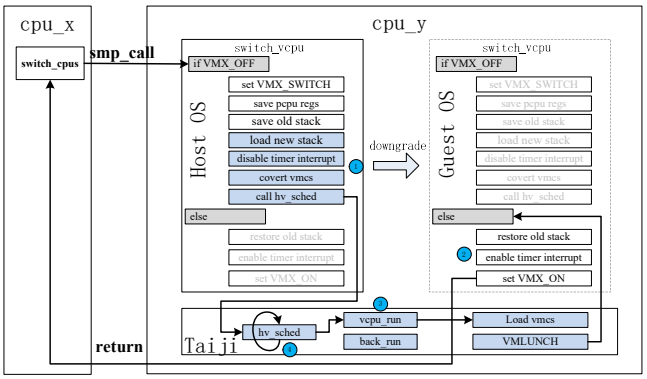


Figure 6. The VMX Switching Flow

usage characteristics of the DPU and manages physical memory mainly at huge page granularity. The current kernel lacks an LRU mechanism for huge pages [41, 45, 52]. Due to the coarse granularity, access to a single base page within a huge page may cause abrupt, transient changes in its hot/cold state that do not reflect actual thermal behavior. To address this, Taiji leverages temporal locality with time-based stabilization to smooth such fluctuations. It employs a multi-level hot/cold set structure with parallel multi-tasking for efficient, accurate cold page identification at huge page granularity. As shown in Figure 7, hot pages are kept in the `hot` set and cold pages in the `cold` set. Pages transitioning between hot and cold are placed in `inactive` and `active` sets. Intermediate sets between `hot` and `active`, and between `inactive` and `cold`, smooth stabilization during periodic scans; if a page's state remains unchanged after a scan, it gradually shifts toward the `hot` or `cold` end. Within each set, elements are ordered by arrival time to indicate relative temperature, e.g., in the `cold` set, the head is colder than the tail. An LRU background task runs per PCPU to periodically scan and manage these sets, accelerating state updates. Each PCPU also maintains a scan cache to buffer scan tasks and results, reducing lock contention. The scheduler ensures LRU tasks execute timely to capture hot/cold state changes.

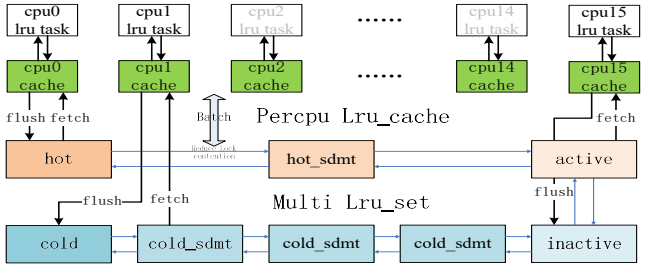


Figure 7. The Multi-level Hot/Cold Sets

4.2 Memory Elasticity

4.2.1 Parallel Multi-Level LRU.

To achieve **O2** of elastic memory, accurate identification of hot and cold states of swapped memory is required. Taiji follows the huge page

4.2.2 Parallel Low-Latency SWAP.

The prerequisite for DPU memory elasticity is preserving native performance, requiring 90% of page fault latencies <10 μs (**O2**). Swap-path latency has two parts: 1) The **backend** choice critically affects

latency: disk or file-based backends cannot meet requirements [23], while keeping swapped data in memory greatly reduces it. Given the prevalence of cold pages on DPUs and the maturity of backend techniques, Taiji uses in-memory zero pages [50] and compression [58], prioritizing zero pages to minimize backend latency. 2) For **frontend** granularity, since the smallest fault unit is a small page, swapping in at huge page granularity causes high latency [16, 57]. However, huge pages reduce performance and metadata overhead from small-page management and avoid allocation failures from fragmentation [14]. To balance this, Taiji manages swapping at memory section (MS, huge page) granularity but operates at memory page (MP, small page) granularity. A huge page is fully swapped only when all its small pages are swapped in or out. Leveraging swap-in/swap-out differences, Taiji parallelizes MP-level swap-ins to meet **O2** efficiency requirements, while swap-outs are sequential to simplify processing.

Thus, **controlling parallel swap-in of MPs within an MS is key to low-latency memory swapping**. Existing techniques lack parallel swap-in support for huge pages, requiring a design to ensure concurrency and atomicity of small-page swaps within them. Taiji defines an MS-level request entity `req` (containing GFN, PFN, locks, bitmaps, MS/MP state) for concurrency control, with three task types: `Fault_in` (passive page-fault–triggered), `Swap_out` (proactive reclamation), and `Swap_in` (prefetch or compaction, omitted here). Concurrency control mainly uses the first two. As shown in Figure 8, four atomicity layers—**req abstraction, read–write locks, execution bitmap, and MS/MP state**—coordinate to ensure efficient low-latency swap-in.

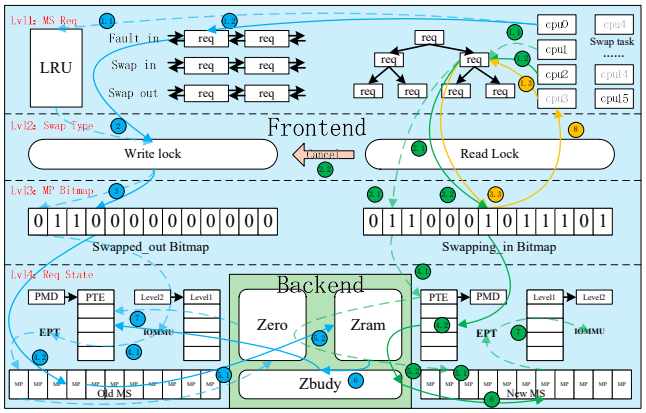


Figure 8. The Swap Parallel

First, `req` operates at MS granularity, each with an independent MS-level lock to prevent contention across MSs and allow parallel swaps for different MSs. All `req`s are unique and stored in a red-black tree for efficient page-fault lookup. On initial swap-out, a new `req` is created from the LRU (1.1); for later active tasks, an existing `req` is fetched from the queue (1.2). Passive swap-ins find the corresponding `req` in the tree by the faulting address (1.1–1.3).

Second, the `req`'s read-write lock enforces mutual exclusion for active swap tasks while allowing passive swaps in parallel. Active tasks, without strict latency needs, are serialized via a write lock to simplify control flow (2). Passive swap-ins require low latency; since multiple MPs within an MS may fault concurrently (2.1), read locks allow parallelism to reduce latency. On read-write lock conflicts, a cancel mechanism ensures the write-locked task exits promptly (2.2).

Third, bitmaps ensure atomicity of swap operations on the same MP within an MS while enabling parallelism across MPs. Two bitmaps are maintained: an already-swapped-out bitmap and a currently-swapping-in bitmap. The first is set during swap-out (3) to ensure swap-in applies only to swapped-out MPs (3.1–3.2). The second guarantees atomicity for page fault–driven swap-ins: if multiple faults hit the same MP, only one proceeds with the swap-in (3.3).

Finally, state control prevents races during critical operations. For swap-out, EPT and IOMMU page tables are split at the first MP's swap-out (4.1) and the MS is reclaimed after the last MP's swap-out. For swap-in, a new MS is allocated at the first MP's swap-in and EPT/IOMMU tables are merged after the last MP's swap-in (7). These operations must execute exactly once at defined state transitions, with parallel flows avoiding races that could cause data corruption.

After frontend concurrency and correctness control, swap-out and swap-in access backend storage for memory writes (5–7) or reads (4–6). These concurrency designs achieve high swap-in performance with low complexity. With coordinated frontend and backend designs, Taiji swaps in pages on EPT page faults within 10µs at the 90th percentile, meeting **O2**. To meet **O3**, Taiji adopts a watermark-based swapping policy, allowing to reserve free memory for DPU memory bursts. Three watermarks are set: `high`, `low`, and `min`. Swapping starts when memory drops below `low` and stops when it rises above `high`. `min` marks critically low memory, triggering proactive swap-out during page faults to avoid prolonged low-memory states. Policies can be tuned, such as halting reclaim between `low` and `high` if no cold pages exist, or starting reclaim below `high` to prepare for sudden demand.

4.3 Resource Scheduler

After resource elasticity, parallel elasticity tasks in Taiji consume CPU, but most DPU CPUs are dedicated to high-performance DPs, leaving only CPs for background tasks. CPs can yield resources to elasticity tasks but still require service guarantees. To avoid impacting native workloads, a scheduler coordinates VCPU tasks for Guest OS workloads and background memory-elasticity tasks. As shown in Figure 9, once the virtualization layer activates and PCPUs are switched to VCPUs, each VCPU becomes a PCPU task managed by the `hv_sched` scheduler in root mode. The scheduler runs in fixed cycles, adjusting background task time slices by CPU load and elasticity urgency, ensuring balanced execution through static configuration and dynamic adjustment.

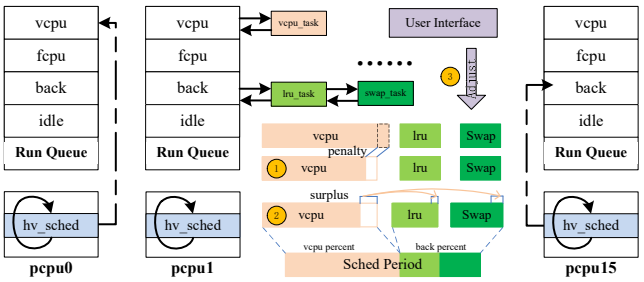


Figure 9. The Scheduler Architecture

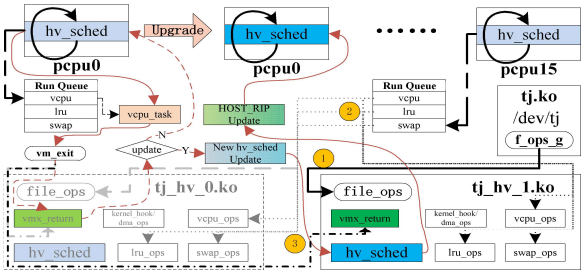


Figure 10. The Upgrade of Taiji.

First, the scheduler prioritizes VCPU execution, running background tasks only when resources permit. Since native DPU workloads must not be affected and prolonged VCPU starvation can hang the Guest OS, static configuration assigns priorities to task types. Each priority level receives a proportional share of execution time in each scheduling cycle based on task urgency. After a PCPU is virtualized, a run queue (`rq`) is created per PCPU with four priority queues: 1) VCPU, active VCPUs after switching; 2) FCPU, reserved for future new VCPU creation; 3) BACK, background tasks such as lru and swap; 4) IDLE, idle tasks. Higher-priority queues get larger execution shares, and tasks within each level share that level's slice.

Second, dynamically, the scheduler tracks each task's start time and enforces a maximum duration. If a task exceeds its limit, a penalty reduces its time slice in later cycles (1). Within a cycle, unused time slices are reallocated to tasks of the same or lower priority (2). Users can adjust the set of CPs allowed for background tasks and their time slices via monitoring tools (3). At the end of each cycle or upon configuration updates, the scheduler recalculates allocations based on updated ratios and cycle period. Proportional allocation and fair adjustments ensure precision across priority levels, enabling efficient elasticity tasks without impacting native DPU workloads.

4.4 Hot Upgrade

To achieve **O4**, Taiji requires hot-upgrade capability in addition to hot-switch. This is challenging, as it must replace the running virtualization logic online. Once active, Taiji's scheduler runs in a loop with prioritized tasks, each priority having its own operation set, and multiple external entry points bound to specific logic that may be invoked anytime. A hot upgrade must seamlessly redirect all old-module operations to the new module at the right moment without disrupting stability.

To this end, Taiji's hot-upgrade solution is designed architecturally to allow updating by simply loading a new module. Based on this modular approach, Taiji is split into two independent parts: `tj.ko` (entry functionality) and `tj_hv_x.ko` (main functionality), as shown in Figure 10. The simple `tj.ko` logic does not require upgrading, while `tj_hv_x.ko`, containing complex logic, does. This allows the underlying complex functions to be upgraded transparently to user services relying on Taiji. The specific upgrade details are as follows:

Data Plane Compatibility: To simplify hot upgrades and avoid data conversion, Taiji's metadata structures must remain compatible so that metadata managed by the old module can be directly inherited. Structure sizes must remain unchanged, with reserved fields for future use. Unless through complex conversion, the semantics and positions of existing fields must not change.

Various Operation Entry Points: The entry module (`tj.ko`) provides a unified file operation entry point. Through `devtj`, each device operation is linked to a file structure whose `f_ops` field points to the global entry `f_ops_g`, which redirects to actions in `tj_hv_x.ko`. For a hot upgrade, only this global entry needs updating, not every open file structure (1). Similarly, functional interfaces (e.g., per-priority scheduler operations, encapsulated `dma_ops`, and externally registered functions) are updated to the new module's addresses (2). All updates occur only after calls to the old module complete.

VCPU Execution Transition: After the VCPU switch, the PCPU cyclically executes the original module's `hv_sched`. During an upgrade, a VCPU may be running in non-root mode and unable to respond promptly to the upgrade request. On Exit, the VCPU returns to the old module's exit location recorded in the `HOST_RIP` field of the VMCS. To handle this, each PCPU maintains an update flag and the new module's loop entry; if set, execution jumps to the new scheduler loop and updates `HOST_RIP` to the new exit location, enabling a seamless handoff (3). When all VCPUs complete this switch, the Taiji hot upgrade is finished.

5 Evaluation

Taiji is implemented in the Linux 4.19 kernel, runs on Intel-based DPUs, and has been deployed at scale over 30,000 units. Performance results show no degradation of DPU performance, confirming virtualization efficiency (Section 5.1). Static and dynamic overhead evaluate its lightweight design for constrained DPUs (Section 5.2). High-load hot-upgrade runtime data and online production statistics further demonstrate the efficiency and benefits of its elasticity (Section 5.3).

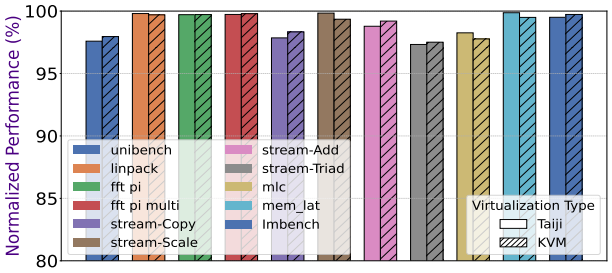


Figure 11. The Normalized Performance of Benchmark.

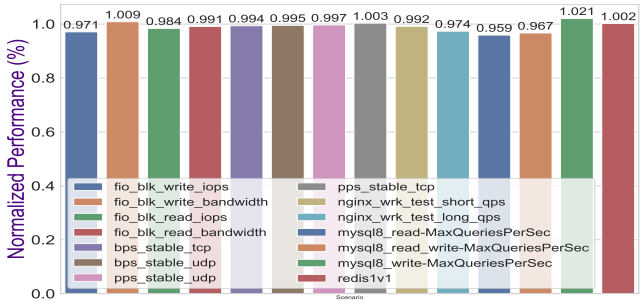


Figure 12. The Normalized Performance of Cloud Workload.

5.1 Performance

The performance test used two pairs of DPU cloud servers. The compute node had an AMD 2.7 GHz 96-core CPU with 1 TB memory. The DPU had an Intel(R) 2.00 GHz 16-core CPU, with one group using a 32 GB DPU plus 16 GB virtual elastic memory, and the other a non-virtualized 64 GB DPU.

5.1.1 Benchmark. We employ a comprehensive set of benchmarks to evaluate Taiji's virtualization overhead, covering CPU performance, memory bandwidth, latency, and cache management. To verify the virtualization efficiency of Taiji, we compared it with KVM. Figure 11 presents the results. Key observations: 1) Unibench: Taiji shows a 2.4% drop, within normal virtualization overhead. 2) Linpack: Performance is nearly identical, overhead negligible. 3) FFTPI: Similar to Linpack, with only minor differences from fluctuation or experimental error. 4) MLC: 1.8% difference, with stable results and minimal overhead. 5) Stream: maximum difference of 2.6%, within acceptable limits. Overall, DPU CPU and memory performance under Taiji virtualization is comparable to native, with total overhead under 3%. Compared to KVM virtualization, both incur similar performance loss. These results indicate minimal impact, meeting **O1**.

5.1.2 Cloud Workload. Figure 12 evaluates Taiji's impact on VM business performance using instance-selling workload tests. 1) Storage: In tests such as `fio_blk`, Taiji achieves performance similar to native; IOPS drop by about 3%, bandwidth is nearly identical, and overhead is within expectations. 2) Network: `bpps` and `pps` show minor fluctuations from random load variations, with performance ratio near 1.0. `nginx` drops slightly, within 3%, indicating minimal loss. 3) MySQL and Redis: `mysql8_read` throughput decreases slightly, within acceptable limits. Overall, compared to native, Taiji introduces no significant difference in core DPU storage and network performance, with memory elasticity overhead acceptable, demonstrating that its memory elasticity and task scheduling meet **O1**.

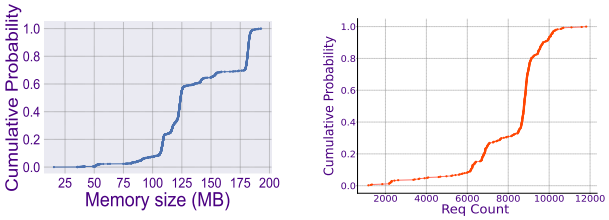
5.2 Lightweightness

5.2.1 Code Size. Table 2 shows the code size distribution of Taiji's modules. All are lightweight; although core modules such as VMX, LRU, and Swap are slightly larger, they are far smaller than their Linux counterparts. VMX, the largest and most complex, has 9,557 LOC, compared to KVM's 77,000 LOC. Memory management modules (MS: 3,273; LRU: 4,202; Swap: 4,101) are minimal relative to Linux's 151,000. The scheduler (Sched) has 2,755 lines, versus Linux's 42,000. Thus, unlike traditional virtualization that adds an entire OS, Taiji remains lightweight in code complexity and does not worsen DPU resource constraints.

Table 2. Code Lines for Each Module in Taiji

MOD	Mpool	MS	VMX	Attr	LRU	Sched	Swap	API
2567	2492	3273	9557	3158	4202	2755	4101	3063

5.2.2 Metadata. Figure 13a shows the CMF of metadata utilization in Taiji's `mpool`. Of the 400 MB reserved, average usage is 127.33 MB (46.69%), peaking below 200 MB; 68.53% is for full pages (EPT and IOMMU page tables) and 31.47% for slab pages (swap, LRU). By minimizing overhead, Taiji uses under one-third of reserved memory, enabling future expansion. Including all reserved metadata, total resource overhead is 1.2% and actual metadata overhead 0.38%, far below the ~5% of conventional virtualization, meeting **O1**.



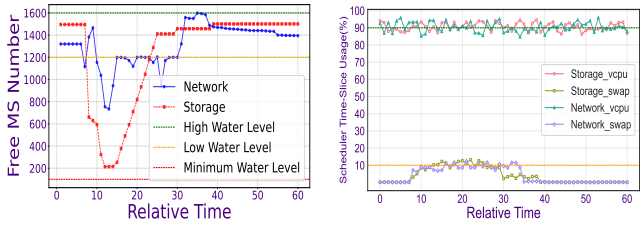
(a) The Metadata Size. **(b)** The Swap *req*

Figure 13. The Running Cost and Benefit of Taiji.

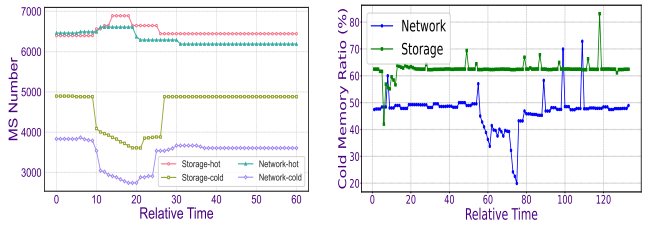
5.3 Memory Elasticity

5.3.1 High-Load Operation Analysis. For high-load scenarios, we test hot upgrades for network and storage, where processes transfer up to 8 GB of data and perform data copying and conversion between old and new processes. As shown in Figure 14e, hot upgrades trigger memory access

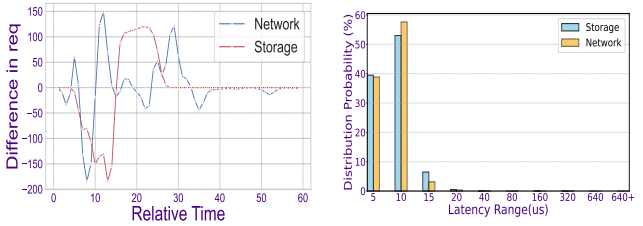
spikes, dropping free memory below the `low` threshold. Taiji proactively swaps until free memory exceeds `high`, adapting to workload changes. Even under high load, memory stays above `min`, preventing exhaustion and ensuring stability. As shown in Figure 14b, during hot upgrade, fair scheduling ensures stable frontend vCPU execution with mostly consistent time-slice allocation. The backend swap task remains idle before upgrade due to meeting watermark thresholds, becomes active during upgrade, and is later constrained to the expected scheduling ratio. These results show Taiji's front-end and back-end scheduling promptly dispatches elastic memory tasks to meet burst demands, achieving **O3**.



(a) Differences in Waterlevel **(b)** Scheduler Slices Ratio



(c) Hot and Cold MS Number. **(d)** Cold Memory Ratio



(e) Differences in Swap *req* **(f)** Latency of EPT Fault

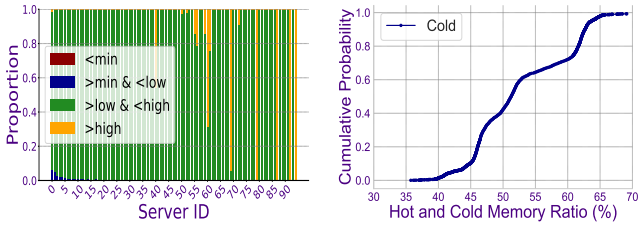
Figure 14. The Running Cost and Benefit of Taiji.

As shown in Figure 14c, the hot page count remains stable during hot upgrade due to multi-level hot/cold tracking that filters transient accesses and prevents abrupt fluctuations. Increased memory access triggers proactive reclamation, reducing cold page count during the upgrade. Figure 14d also shows cold data changes: it drops sharply as memory is swapped in and added to the `hot` set before stabilizing, with minor upward spikes from LRU cache refreshes. Storage stabilizes faster than network due to lower memory use, causing fewer faults and swaps. Even under intensive access, substantial reserved memory remains: for network, the cold-page ratio briefly drops to 20% but stays above 35%; for storage, it remains above 40%. These results show that even under high load, cold pages can sustain DPU memory elasticity.

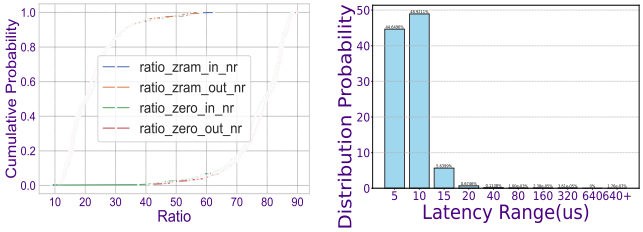
Figure 14e shows `req` variation during hot upgrades as the difference from pre-upgrade values. Fluctuations are generally small. In network upgrades, an initial memory access spike causes heavy swap-in, lowering `req`; proactive swap-out then raises it, with smaller oscillations reflecting multiple intensive access phases, all within acceptable limits. In storage, `req` also drops initially due to large transfer memory use, triggering the water-level policy and proactive swap-out, after which `req` rises and stabilizes once demand is met. Unlike network, storage triggers the policy only once, indicating more concentrated access.

Figure 14f shows the EPT fault latency distribution during hot upgrades. Under large-scale memory access, over 90% of EPT page fault latencies are below 10 μs, with 92.51% for storage and 95.50% for network. No significant long-tail latency is observed. This shows that even under extreme load, Taiji sustains low swap-in latency, achieving **O2**.

5.3.2 Online Operation Analysis. To evaluate Taiji's practical behavior, we analyze runtime memory swapping using real online data. Each DPU has 32 GB physical and 16 GB virtual memory, achieving 50% elastic expansion. Figure 15a shows memory water levels across cluster nodes over one hour: most remain between `low` and `high`, some above `high`, and only a few occasionally below `low` due to memory-intensive workloads like hot upgrades or traffic spikes. This indicates most nodes maintain safe levels, ensuring burst capacity and demonstrating Taiji's water-level policy sustains diverse production workloads.



(a) Waterlevel Distribution **(b)** Cold Memory Ratio



(c) Swap-in/Swap-out Ratio **(d)** Latency of EPT Fault

Figure 15. The Running Cost and Benefit of Taiji.

Figure 15b shows the CMF of the cold/hot memory ratio, indicating Taiji's multi-level LRU accurately identifies cold pages. The cluster's average cold-memory ratio is 52.79%, and even the most memory-utilized nodes stay above 30%, ensuring ample swap space for DPU expansion. Figure 15c shows that of all swapped MPs, 76.79% are zero pages and

23.21% are compressed pages, with an average compression ratio of 47.63%, greatly reducing their footprint. These results show Taiji's elastic back-end memory reduces space overhead and improves overall memory utilization.

Figure 15d shows the percentile distribution of EPT latencies across nodes during this period: 99% are below 15 μs, and 93.57% are within 10 μs, meeting **O2**. This confirms that Taiji's page-fault latency distribution across the cluster meets design expectations, enabling timely swap-in to minimize application performance impact. Occasional higher latencies indicate room for improvement in EPT fault handling, which future optimizations should address.

5.3.3 Overcommitting Benefit. Taiji supports multiple elastic memory configurations, with the common case being a 32 GB DPU with 16 GB virtual elastic memory, achieving 50% elasticity. Figure 13b shows the CMF of swapped pages: about 50% of nodes swap fewer than 8,000 MSes, and most fall between 6,000 and 11,000. Swapping 8,000 MSes frees 15.6 GB of memory. The `Req` metadata overhead for recording swap information is about 20 MB, including 15 MB for CRC values to ensure correctness. From the compression analysis in Section 5.3.2, storing these swapped pages uses only 1.73 GB, yielding a 9× overselling gain. Compared to the average metadata cost of 127.33 MB or a reserved 400 MB, the benefit-to-cost ratio reaches 125.5× or 39×, fully meeting **O3**.

6 Related Work

With rising demands in cloud computing, big data, and AI, companies like AWS [3], NVIDIA [5, 39], and Intel introduced DPUs [9] to meet data center network/storage needs [9, 21, 38]. NVIDIA's BlueField integrates ARM cores, programmable accelerators, and high-speed interfaces, offloading IO from CPUs to boost performance [38]. Programmability supports diverse workloads; in clouds, DPUs reduce host CPU load and improve efficiency [29]. Yet DPU performance faces challenges: complex hardware and slow upgrades limit adaptability [9, 13]. Growing memory/CPU demands and slow upgrade cycles make elasticity urgent. Existing techniques target cloud services but neglect DPU-specific needs [7, 35]. With many DPUs deployed, hot-swappable elasticity is needed.

Memory overcommitment allocates virtual machines more memory than the host's physical capacity to optimize resource use [10]. If mismanaged, it can degrade performance [36], and reclamation techniques such as ballooning cause slowdowns from increased paging activity and workload variability [4]. While ESXi[15], KVM, and Xen support overcommitment, vendors discourage its use to avoid unpredictable performance loss [18, 59]. Double paging by both hypervisor and guest OS adds severe overhead, and in containerized environments, overcommitment also significantly impacts performance [36]. Substantial improvements are required for elastic memory to be practical at production scale.

Researchers proposed methods for memory overcommitment: MemFlex [65] enhances guest swapping; Zweilous [27] improves compression; PMA [61] reclaims inactive memory with <10% loss and 33% overcommitment; HeMem [43] manages tiered memory asynchronously; Pond [28] builds CXL pools with ML-based prediction. Application-aware approaches [19, 45] also maintain acceptable performance. TMO [58] measures work loss and adjusts offloading in real time. However, in high-performance DPUs, overcommitment still suffers from unpredictable degradation and complex management [61]. Thus, lightweight, scalable, high-performance virtualization for DPUs is urgently needed.

7 Discussion

7.1 Data Correctness

A key factor in DPU performance is extensive DMA usage, but as current DMA devices lack retry support, swapping memory must be avoided to prevent corruption. Binding all device-used I/O memory leaves too little movable memory to expand virtual memory by over 50% (**O3**). Since this DMA-related memory is dynamic, Taiji lets applications specify DMA ranges for protection and ensures timely swap-in before access. Taiji also intercepts DMAR exceptions and uses CRC to ensure correctness. Future hardware–software co-design will enable DMA-triggered swap-ins without application tags. These mechanisms allow Taiji to match **O2**'s efficiency without compromising data correctness.

7.2 Backend Memory

Taiji's backend storage supports zero, compressed, free pages, remote memory, and disks. In the guest OS, it detects swapped free pages and reclaims their page tables immediately, with later swap-in only rebuilding tables—enhancing efficiency and aiding **O2**'s low-latency goal. As such pages are rare and detection incurs zone lock overhead that disrupts the guest OS, this type is disabled in production. Remote memory and disks act as fallback storage for bursts beyond Taiji's elasticity, though none have occurred; the common case is hot upgrades temporarily doubling memory, which current swapping handles. These backends remain optional.

7.3 Portability

Taiji is currently implemented on DPUs with Intel's x86 CPU, whose virtualization is closely tied to the underlying hardware, while elastic memory is hardware-agnostic. By adapting hardware-specific features and reusing other components, Taiji can be ported to other virtualization platforms such as AMD and ARM. For AMD, adaptations include differences between SVM and VMX, IOMMU formats, LAPIC, and timer control. Once addressed, CPU virtualization can run smoothly. Porting to ARM is similar but requires additional

work due to substantial differences in registers and virtualization modes from x86. Resolving these hardware differences enables Taiji to be ported to other DPU platforms.

7.4 Extensibility

In scheduling, CPs yield capacity to elastic tasks, but under high load or growing demands, resource shortages may occur. Taiji's extensible scheduler can create VCPUs on demand by initializing a VMCS without register passthrough. When hot-plugging a VCPU, the virtualization layer detects its startup signal and, with the scheduler allocating time slices, schedules the VCPU task. The new VCPU behaves like a switched VCPU and, with proper time slices, runs normally in the virtualization layer, transparently adding CPU resources for elastic scaling. Coordinating and fully utilizing physical CPUs via the scheduler, Taiji unifies memory and CPU elasticity while reducing maintenance cost.

8 Conclusion and Future Work

We present Taiji, a lightweight DPU elasticity solution that inserts a virtualization layer between OS and hardware, enabling full memory swapping and elastic scaling. Leveraging multi-level LRUs, parallel swapping, and transparent scheduling, Taiji achieves efficient memory elasticity. Its modular design supports hot switching and upgrades, improving availability on existing and new DPUs. Experiments show Taiji improves utilization, lowers hardware cost, and raises service capacity while sustaining high performance. Future work includes porting beyond x86, enhancing CPU elasticity, and eliminating user-space DMA tagging.

References

[1] 2025. HugeTLB pages cannot be swapped. Linux Kernel Documentation. https://docs.kernel.org/admin-guide/mm/hugetlbpage.html "Pages that are used as huge pages … cannot be swapped out under memory pressure.".

[2] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 631–644.

[3] Amazon Web Services. 2024. *The Security Design of the AWS Nitro System*. Technical Report. AWS. Whitepaper.

[4] Nadav Amit, Dan Tsafrir, and A. Schuster. 2014. VSwapper: a memory swapper for virtualized environments. *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*.

[5] Darren Ng Xiaoyi Lu Arjun Kashyap, Yuke Li. 2025. Understanding the Idiosyncrasies of Emerging BlueField DPUs. *Proceedings of ICS '25* (2025). Salt Lake City, UT, USA.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *Operating Systems Review* (2003).

[8] Shai Bergman, Mark Silberstein, Takahiro Shinagawa, Peter Pietzuch, and Lluís Vilanova. 2023. Translation Pass-Through for Near-Native Paging Performance in VMs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 753–768.

[9] Idan Burstein. 2021. Nvidia Data Center Processing Unit (DPU) Architecture. *2021 IEEE Hot Chips 33 Symposium (HCS)* (2021), 1–20.

[10] Chia-Hao Chang, Jihoon Han, Anand Sivasubramaniam, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-Mei Hwu. 2024. Gmt: GPU orchestrated memory tiering for the big data era. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS)*. 464–478.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 273–286.

[12] Alibaba Clouder. 2019. Alibaba Cloud Announces 3rd Generation of X-Dragon Architecture. (2019). https://www.alibabacloud.com/blog/alibaba-cloud-announces-3rd-generation-of-x-dragon-architecture_595397

[13] Jaideep Dastidar. 2023. FPGAs and Their Evolving Role in Domain Specific Architectures: A Case Study of the AMD 400G Adaptive SmartNIC/DPU SoC. *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (2023).

[14] Johannes K Fichte, Norbert Manthey, Julian Stecklina, and André Schidler. 2020. Towards faster reasoners by using transparent huge pages. In *International Conference on Principles and Practice of Constraint Programming*. Springer, 304–322.

[15] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Istanbul, Turkey, March 14-15, 2015*. ACM, 39–51. https://doi.org/10.1145/2731186.2731187

[16] Kaijie Guo, Dingji Li, Ben Luo, Yibin Shen, Kaihuan Peng, Ning Luo, Shengdong Dai, Chen Liang, Jianming Song, Hang Yang, et al. 2024. VPRI: Efficient I/O Page Fault Handling via Software-Hardware Co-Design for IaaS Clouds. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 541–557.

[17] Edward Haletky. 2011. *VMware ESX and ESXi in the Enterprise: Planning Deployment of Virtualization Servers*. Pearson Education.

[18] Red Hat. 2024. Overcommitting with KVM. (2024). https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/chap-overcommitting_with_kvm

[19] M. Hines, Abel Gordon, Márcio Silva, D. Silva, K. Ryu, and Muli Ben-Yehuda. 2011. Applications Know Best: Performance-Driven Memory Overcommit with Ginkgo. *2011 IEEE Third International Conference on Cloud Computing Technology and Science* (2011), 130–137.

[20] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 588–604.

[21] Intel. 2024. Introduction to Intel IPUs. (2024). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html

[22] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, and Xiaoning Ding. 2023. Making dynamic page coalescing effective on virtualized clouds. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 298–313.

[23] Insoon Jo. 2023. Toward Ultra-Low Latency SSDs: Analyzing the Impact on Data-Intensive Workloads. *Electronics* 13, 1 (2023), 174.

[24] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. Dttawa, Dntorio, Canada, 225–230.

[25] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 317–330.

[26] Dingji Li, Zeyu Mi, Baodong Wu, Xun Chen, Yongwang Zhao, Zuohua Ding, and Haibo Chen. 2021. Accelerator Virtualization Framework Based on Inter-VM Exitless Communication. *Int. J. Softw. Informatics* 11, 2 (2021), 169–193.

[27] Guoxi Li, Wenzhi Chen, and Yang Xiang. 2021. Zweilous: A Decoupled and Flexible Memory Management Framework. *IEEE Trans. Computers* 70, 9 (2021), 1350–1362.

[28] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 574–587.

[29] Yuke Li, Arjun Kashyap, Yanfei Guo, and Xiaoyi Lu. 2023. Characterizing Lossy and Lossless Compression on Emerging BlueField DPU Architectures. *2023 IEEE Symposium on High-Performance Interconnects (HOTI)* (2023), 33–40.

[30] Haiying Shen Zhongyi Zhou Liuhua Chen, Shilkumar Patel. 2015. Profiling and Understanding Virtualization Overhead in Cloud. In *ICPP (2015)*. Examines overhead from hypervisor trapping, CPU scheduling, I/O virtualization.

[31] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1–18.

[32] Richard McDougall and Jennifer Anderson. 2010. Virtualization performance: perspectives and challenges ahead. *SIGOPS Oper. Syst. Rev.* 44, 4 (Dec. 2010), 40–56.

[33] Dmitry Sepp Adamos Ttofari Leon Thomm Do Le Quoc Siddharth Chandrasekaran Sharan Santhanam Chuan Ye Shai Bergman Wei Wang Sven Lundgren Konstantinos Sagonas Alberto Ros Milan Pandurov, Lukas Humbel. 2024. Flexible Swapping for the Cloud. *arXiv preprint arXiv:2409.13327* (2024). "HugeTLB swapping is not supported by Linux, and THP backed memory will be split into 4 kB pages".

[34] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2024. Enoki: High velocity linux kernel scheduler development. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 962–980.

[35] Roberto Morabito, Jimmy Kjällman, and Miika Komu. 2015. Hypervisors vs. Lightweight Virtualization: A Performance Comparison. *2015 IEEE International Conference on Cloud Engineering* (2015), 386–393.

[36] Rina Nakazawa, Kazunori Ogata, Seetharami Seelam, and Tamiya Onodera. 2017. Taming Performance Degradation of Containers in the Case of Extreme Memory Overcommitment. *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)* (2017), 196–204.

[37] Matthias Nickel and Diana Göhringer. 2024. A Survey on Architectures, Hardware Acceleration and Challenges for In-Network Computing. *ACM Transactions on Reconfigurable Technology and Systems* (2024).

[38] NVIDIA. 2021. NVIDIA BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit

[39] NVIDIA Developer Blog. 2021. Offloading and Isolating Data Center Workloads with NVIDIA BlueField DPU. https://developer.nvidia.com/blog/offloading-and-isolating-data-center-workloads-with-bluefield-dpu/.

[40] Ashish Panwar, Aravinda Prasad, and K Gopinath. 2018. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 679–692.

[41] SeongJae Park. 2025. DAMON: Kernel Subsystem for Data Access Monitoring and Access-aware System Operations. https://damonitor.github.io/.

[42] Michael Pearce, Sherali Zeadally, and Ray Hunt. 2013. Virtualization: Issues, security threats, and solutions. *ACM computing surveys (CSUR)* 45, 2 (2013), 1–39.

[43] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 392–407.

[44] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 803–817.

[45] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2023. vTMM: Tiered Memory Management for Virtual Machines. *Proceedings of the Eighteenth European Conference on Computer Systems* (2023).

[46] Sai Sha, Yi Zhang, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2021. Swift shadow paging (SSP): No write-protection but following TLB flushing. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 29–42.

[47] Nimish Shah, Wannes Meert, and Marian Verhelst. 2023. DAG Processing Unit Version 1 (DPU): Efficient Execution of Irregular Workloads on a Multicore Processor. In *Efficient Execution of Irregular Dataflow Graphs: Hardware/Software Co-optimization for Probabilistic AI and Sparse Linear Algebra*. Springer, 69–88.

[48] Nimish Shah, Laura Isabel Galindez Olascoaga, Shirui Zhao, Wannes Meert, and Marian Verhelst. 2021. DPU: DAG processing unit for irregular graphs with precision-scalable posit arithmetic in 28 nm. *IEEE Journal of Solid-State Circuits* 57, 8 (2021), 2586–2596.

[49] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. 2024. Burstable Cloud Block Storage with Data Processing Units. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 783–799.

[50] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. 2016. Efficient memory-mapped I/O on fast storage device. *ACM Transactions on Storage (TOS)* 12, 4 (2016), 1–27.

[51] Bijan Tabatabai, James Sorenson, and Michael M Swift. 2024. FBMM: Making Memory Management Extensible With Filesystems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 785–798.

[52] The Linux kernel documentation. 2025. Multi-Gen LRU: An alternative LRU that optimizes page reclaim and improves performance under memory pressure. Linux kernel admin guide (mm/multigen-lru). https://docs.kernel.org/admin-guide/mm/multigen_lru.html Online documentation.

[53] Meghana Thiyyakat, Subramaniam Kalambur, and Dinkar Sitaram. 2020. Improving resource isolation of critical tasks in a workload. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 45–67.

[54] Anthony Velte and Toby Velte. 2009. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc.

[55] Laurens Versluis, Mehmet Çetin, Caspar Greeven, Kristian B. Laursen, Damian Podareanu, Valeriu Codreanu, Alexandru Uta, and Alexandru Iosup. 2021. A Holistic Analysis of Datacenter Operations: Resource Usage, Energy, and Workload Characterization - Extended Technical Report. *ArXiv* abs/2107.11832 (2021).

[56] VMware. 2024. *Performance Best Practices for VMware vSphere 8.0 Update 3*. https://www.vmware.com/docs/vsphere-esxi-vcenter-server-80U3-performance-best-practices Describes memory overhead and system-wide swap file usage for ESXi.

[57] Yaohui Wang, Ben Luo, and Yibin Shen. 2023. Efficient memory overcommitment for I/O passthrough enabled VMs via fine-grained page meta-data management. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 769–783.

[58] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. 2022. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 609–621.

[59] Xen. 2008. Xen 3.3 Feature : Memory Overcommit. (2008). https://xenproject.org/2008/08/27/xen-33-feature-memory-overcommit/

[60] Yuhan Yang Rong Chen Xingda Wei, Rongxin Cheng and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 987–1004.

[61] Lulu Yao, Yongkun Li, Jiawei Li, Weijie Wu, and Yinlong Xu. 2021. Progressive Memory Adjustment with Performance Guarantee in Virtualized Systems. In *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*. ACM, 86:1–86:11.

[62] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the twenty-third acm symposium on operating systems principles*. 203–216.

[63] Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, and Haibing Guan. 2020. GiantVM: a type-II hypervisor implementing many-to-one virtualization. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 30–44.

[64] Jiyuan Zhang, Weiwei Jia, Siyuan Chai, Peizhe Liu, Jongyul Kim, and Tianyin Xu. 2024. Direct Memory Translation for Virtualized Clouds. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS)*. 287–304.

[65] Qi Zhang, Ling Liu, Gong Su, and Arun Iyengar. 2017. Memflex: A shared memory swapper for high performance vm execution. *IEEE transactions on Computers* 66, 9 (2017), 1645–1652.

[66] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. 2019. Fast and scalable VMM live upgrade in large cloud infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 93–105.

[67] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-density Multi-tenant Bare-metal Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 483–495. https://doi.org/10.1145/3373376.3378507

[68] Zongpu Zhang, Jiangtao Chen, Banghao Ying, Yahui Cao, Lingyu Liu, Jian Li, Xin Zeng, Junyuan Wang, Weigang Li, and Haibing Guan. 2024. Hd-iov: Sw-hw co-designed i/o virtualization with scalability and flexibility for hyper-density cloud. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 834–850.