HyperTier: Guest-Empowered Tiered Memory for Virtualized Clouds with Isolation, Elasticity and Efficiency

Abstract

We present HyperTier, the first hypervisor-based design that transforms and further advances operating system level tiered memory management into elastic and scalable solutions for the virtualized cloud. The main insight behind HyperTier is that, 1) the wrongful assumption of guests' incapacity in managing memory tiers, 2) the overlooked isolation, elasticity problems, in conjunction with 3) the host-and-guest-side management overhead, have hindered the introduction of tiered memory to the cloud, missing out the opportunity of memory scalability growth and total cost of ownership reduction. To address these issues, we first show a generic and modular conversion mechanism through HyperFlex, enabling existing Linux derivatives and purposely built operating systems to adapt to virtualization. And then we further optimize all stages of guest tiered memory management with HyperPlace to cutdown wasted cycles and convert them into rentable CPU resources. We evaluate all transformed and our optimized designs on two kinds of tiered memory configurations, composing of DRAM with either real Persistent Memory or emulated CXL.mem. The results show that the conversion along yields 74% performance improvements for the existing best-performing guest design; our optimized design then boosts performance up to another 90% on real-world workloads co-running in multiple virtual machines compared to the second best alternative.

1 Introduction

1.1 Expand Main Memory Space with Tiered Memory

Major cloud adopters, such as Azure [51], Google [16], and Facebook [46], have turned to tiered memory architectures to expand memory capacity and lower Total Cost of Ownership (TCO). Virtualized clouds typically choose a Virtual Machine (VM) as the primary service offering, often configured with 4 to 32 GiB of memory per core [5, 18, 36]. Yet, while CPU cores have grown tenfold (from 15 to 144 per package [23, 24]), maximum possible memory per CPU thread has dropped to about one-third of its original capacity (from about 50 GiB to roughly 14.2 GiB), limiting future scalability.

Adding a second memory tier of alternative media or interconnects offers a scalable and cost-effective solution to facilitate cloud scalability growth and the offering of larger memory configurations. Technologies like Persistent Memory (PMEM) and Compute Express Link Memory Protocol (CXL.mem) can double memory at half the cost of additional DRAM [14, 15]. Reusing decommissioned DRAM with CXL.mem expansion cards further extends capacity without adding new memory modules [51].

However, these new media introduce higher latency and lowering overall performance. PMEM can be about three times slower than DRAM [50], and CXL.mem about twice as slow [33]. We label them as the slow memory (SMEM) tier and rely on fast memory (FMEM) tier composing of DRAM for speed. By placing frequently accessed (hot) data in DRAM and less-accessed (cold) data in SMEM, we minimize slowdowns. This approach ensures most accesses remain fast, limiting the performance impact of the slower tier.

1.2 Isolation, Elasticity, and Efficiency Requirements

Cloud computing relies on consolidating often underutilized servers into virtual machines on shared hardware, and turns reduced resource wastage into profits. Each VM is designed to appear as if it is running on a dedicated server with a fixed amount of allocated memory. Maintaining such illusion of exclusive hardware access while exploiting the multiplexing opprotunity requires the hypervisor to provision memory in an isolated and elastic manner.

Isolation ensures that tenants' sensitive information, including memory access patterns, remains confidential, not just from other tenants but also from the hypervisor itself. This requirement conflicts with traditional hypervisor-based tiered memory management systems [19, 22, 28, 40, 51], which need to obtain guest VMs' memory access information to optimize data placement across memory tiers.

At the same time, tenants' memory needs naturally fluctuate. This variability provides hypervisors with opportunities to repurpose unused memory when demand decreases and challenges them to reallocate memory resources when demand increases. Elastic memory provisioning is typically achieved using a memory balloon device [45]. However, existing hypervisor-based tiered memory management systems have not yet effectively addressed this issue, often leaving discussions of ballooning mechanisms [28] at a theoretical level [40].

Moreover, the more CPU resources devoted to managing tiered memory, the fewer are available for renting out as VMs, which increases the total cost of ownership. Existing systems consume significant CPU cycles to monitor [40] or replicate guest page tables [19, 28] solely for tracking memory access information and determining data hotness. Combined with the challenges of maintaining isolation and elasticity, these shortcomings motivate us to explore a new approach for integrating tiered memory and the its expansion opportunity into virtualized cloud environments.

1.3 Our Work and Contribution

We analyzed emerging OS–level tiered memory solutions [4, 29, 32, 35, 47, 49] in Section 2.1 and observed that their blooming developments challenge the traditional role of the hypervisor in managing tiered memory.

Contribution 1. We propose a novel perspective that separates data placement responsibilities from tiered memory provisioning. We advocate that hypervisors should focus on isolated and elastic tiered memory provision, while granting guest operating system the autonomy of data placement leveraging their favoured position and awareness to access locality without compromising isolation.

Contribution 2. We introduce a novel elastic tiered memory provisioning solution called HyperFlex enabling the conversion of OS-level tiered memory management systems into a virtualization ready cloud-level solution. HyperFlex presents tiered memory as NUMA nodes to guest operating systems and supports elastic inflation and deflation to adjust both total and per-tier memory allocations. Integrating HyperFlex enables existing OS-level tiered memory management systems to function effectively in virtualized cloud environments without modifications.

However, our evaluation of existing OS-level designs in Figure 1 reveals that they still suffer from significant management overhead, which might scale exponentially with the number of parallel running VMs. In multi-tenant cloud environments, where many VMs run in parallel, the compound effect on management overhead leads to excessive CPU usage, negating the cost benefits of tiered memory. We observe that existing OS-level designs either fail to accurately identify frequently accessed hot data or incur unacceptable overhead due to hint-fault-based promotion and split-LRU maintenance.

Contribution 3. We present a set of optimized designs called HyperPlace for all three stages of tiered memory management. HyperPlace classifies data hotness using samples from virtualization-friendly PMU sampling and a segment tree–based range classification algorithm. It performs symmetric exchanges between the hottest and coldest data after fitting access distribution across ranges to tiered memory capacity.

Contributin 4. We evaluate our tiered memory provisioning mechanism, HyperFlex, and compare converted OS-level management systems with our optimized design across two tiered memory configurations: one with DRAM as fast memory and real PMEM as slow memory, and another with DRAM and emulated CXL.mem. We test these configurations using seven real-world workloads spanning databases, scientific computing, graph processing, and machine learning. Our results show that HyperFlex outperforms traditional balloon-based elastic memory provisioning by 74%, and HyperPlace improves real-world workload performance by up to 90% compared to the next best alternative.

Contributin 5. We open-source our implementation on GitHub at together with all comparison targets, workloads, and utilities needed to reproduce our evaluation results.

2 Background

2.1 Tiered Memory Management

Tiered memory management systems [4, 16, 19, 22, 28, 29, 32, 35, 38, 40, 44, 46, 47, 49, 51] have attracted significant interest with the adoption of PMEM and CXL.mem in real world datacenters [33]. While some systems are designed for single user-space applications [38] or entire data centers [16, 46], most opt for application-transparent management within the operating system kernel [4, 29, 32, 35, 47, 49] or hypervisor [19, 22, 28, 40, 51].

These systems traditionally consist of three components: access sampling, hotness classification, and data migration. They leverage sampled memory accesses to estimate which pages are likely to be frequently accessed in the future, i.e. a process known as hotness classification, and assign these pages to FMEM through data migration.

Access Sampling. Memory accesses are automatically tracked using the Page Table Entry Access (PTE.A) bit by the CPU's Memory Management Unit (MMU) across various architectures. Due to the readiness of PTE.A bits, this is the most common method for access sampling in operating system kernels and hypervisors. However, PTE.A bits have three main drawbacks: 1) they provide only binary information of accessed or not accessed, 2) it's challenging to decide when to collect this information, and 3) collecting it can be difficult and costly.

Converting binary access information into classifiable access frequency requires aggregating results from two [35, 47, 49] or more [29] sampling rounds. In each round, PTE.A bits from the memory regions of interest are scanned and reset to detect accesses in the current round and enable next round's detection. Nimble [49], TPP [35] and Nomad [47] follows the Linux native deisgn, which maintains a *referenced* flag for each page to indicate whether a PTE.A bit is found within the last round. By combining this with the current round's information, they can perform hotness classification. AutoTiering [29] extends each page's *referenced* flag into an 8-bit vector, allowing finer classification.

Ideally, PTE.A bits for each physical page shall be collected at predetermined intervals to reliably transform access information into frequency data. However, current Linux-based systems [29, 35, 47, 49] collect PTE.A bits only during memory reclamation when the free memory is under pressure. The collection process involves scanning every PTE.A bit for a large amount of physical pages. Making things worse, one single physical page might often have different mappings scattered across different processes' page tables. Linux maintains and walks a special reverse mapping *rmap* data structure to locate each Page Table Entry (PTE) that maps a given physical page. Compared to simply following page tables and extracting one PTE.A, this method is more cache-unfriendly. This process also often requires TLB flushes after resetting PTE.A bits.

HeMem [38] first introduced Performance Monitoring Unit (PMU) sampling to collect access samples, with subsequent adaptations in Memtis [32], vTMM [40] and Colloid [44]. Unlike the proactive collection required for PTE.A bits, PMU sampling can generate a continuous stream of load/store access samples after a one-time configuration. PMU sampling is implemented in the CPU core's PMU hardware, which tracks user-selected hardware events. When the PMU event counter overflows, the CPU hardware forms an architectural state sample and writes it to a memory buffer. The PMU counter also supports automatic resets to allow continuous generation of samples.

The highly accurate nature of PMU sampling has led to increased adoption but also raised some debates about unintended overhead during high-frequency sampling [35, 47]. TPP only used PMU sampling during offline performance analysis, but chose not to integrate it into their final design, citing standardization issues among vendors and the tradeoff between sampling frequency and performance, similar to the decision in Nomad. Memtis moved HeMem's userspace PMU sampling into the kernel space. None of the prior work extended PMU sampling into virtualized environments, either due to isolation concerns raised by vTMM [40] or compatibility issues mentioned by Memstrata [51].

Hotness Classification. After identifying which pages have been accessed, hotness classification algorithms select the most frequently accessed ones. Linux derivatives, including Nimble, TPP, and Nomad all preserve Linux's original split-LRU design, which contains two lists, active and inactive lists. During an active list rotation, any page not referenced by PTE.A bits is moved to the inactive list. When rotating the inactive list, if still no PTE.A bits are found for a page, it can be considered for demotion if it is in FMEM. Conversely, if any PTE.A bits are found, the page remains on the inactive list. Previous round results are stored in the *referenced* flag of the page metadata. If a page's PTE.A bits are found in two consecutive rounds, it is moved to the active list.

However, being on the active list does not necessarily imply future migration decisions. Demotion candidates are selected only from the inactive list, and not all pages from the active list will be promotion candidates. Instead of the *referenced* flag, AutoTiering [29] storing up to eight rounds of history in an 8-bit vector, and aggregates them into frequency information. Demotion candidates are selected as the least frequently accessed among the last eight rounds.

Thermostat classifies pages into hot or cold based on a static threshold. Following HeMem, Memtis stores each page's sampled access count in the page's metadata, placing this metadata alongside a page table extension. Unlike Thermostat, Memtis proposes dynamic adaptation of the hot threshold to fit as many pages as possible into fast memory. The hot threshold is chosen to be the largest one before overflowing fast memory. Memtis repurposes the split-LRU's active and inactive lists to store promotion and demotion candidates. When a page's access frequency meets or exceeds the hot threshold, it is placed onto the active list; otherwise, it is placed onto the inactive list.

Data Migration. Data migration moves identified hot pages to fast memory and cold pages to slow memory. Linux derivatives handle promotions and demotions with different strategies. Promotions are handled semi-proactively through NUMA hinting faults, introduced by the AutoNUMA patch [7]. In AutoNUMA, parts of a process's virtual space are unmapped during a NUMA scan period. Subsequent accesses to these pages trigger NUMA hinting faults, allowing the kernel to discover pages not located on the local NUMA node and migrate them.

This mechanism is inherited by Linux derivatives to implement promotion. Nimble added support for migrating huge pages. Memtis found that such promotion policies are on the application's critical memory access path. To mitigate this, Memtis adopted HeMem's threaded design, introducing a dedicated promotion thread to offload data movements to the background. Nomad addresses the critical path problem differently by introducing shallow copies of memory pages. Promotions found in hint faults are processed in the background, and applications are allowed to continue accessing slow memory until the background copy completes. Inconsistencies during promotion are addressed through retries, and the old slow memory copies are kept as shadow copies.

Linux passively performs demotion as part of the memory reclamation procedure, triggered when memory allocation fails due to insufficient free space. Demoting pages to slow memory frees up space to satisfy allocation requests. TPP improved on this by checking the amount of free memory during every task switch and triggering demotion when it falls below a *demote watermark*. AutoTiering optimized the demotion path for a four-tiered system. If the demotion involves a previously promoted page in Nomad, the original slow memory shadow copies can be reused through a page table remapping. However, keeping the shadow copies in sync with the fast memory master copies requires triggering a page fault on every write to the master copies. Nomad only keeps the shadow copies for clean master copies with no writes; the shadow copies are discarded on the first write that triggers a page fault. Memtis used a similar watermark-oriented demotion policy as TPP but carried out the demotion process in a background thread rather than integrating it with the scheduler.

2.1.1 PMU Sampling Support under Virtualization

Contrary to prior assumptions [40, 51], we find that PMU sampling is now well-supported with strong privacy guarantees for guest states under virtualization [48]. Even crossarchitecture support for PMU sampling exists [37], allowing extending PMU sampling based hotness classification to a wide arrange of machines in the cloud with simple changes on the collected events. Our testbed is an Intel-based machine as introduced in Section 3.2. We focus on Intel's implementation of PMU sampling, known as Precise Event-Based Sampling (PEBS) [26], in the following discussion.

The main concern for guest PEBS support is the *potential* breach of isolation boundaries caused by sharing the sample buffer and leaking sensitive information across VMs. Prior hypervisor-based management systems intuitively assume that PEBS does not support virtualization [51] and PEBS enabled in the guests will generate samples and write to the host-side buffer [40] which is shared across all VMs, leaking load/store addresses via generated samples.

However, we found that this is not the case. PEBS is both virtualization-ready and generates samples in a well isolated manner. The PEBS sample buffer is part of the CPU debug control data structure and is controlled via a debug control register. Intel's hardware virtualization supports automatic switching to a guest-private debug control data structure upon VM entry through the Virtual Machine Control Structure's `vmcs.debugctl` field, making the sample buffer guest-private as well. Samples generated in executing different VMs will be written to its private buffer. Although the `vmcs.debugctl` field is hypervisor-visible, it contains only a Guest Virtual Address (GVA), preventing the hypervisor from accessing the actual location and content of the sample buffer. Programming the PMU hardware also involves only guest virtual addresses. The guest programs the PMU debug control register through Model-Specific Register (MSR) writes, using `wrmsr` instructions, just as it would on bare metal. These writes trap into the hypervisor and are redirected to the `vmcs.debugctl` field.

Guest PEBS support has existed since early versions of PEBS [26]. However, little attention was paid to it due to an old bug not fixed until version 5 [48]. Guest memory pages are usually lazily allocated upon Extended Page Tables (EPT) page faults, allowing the hypervisor to overcommit memory beyond the total physical capacity. In early PEBS implementations, the sample write process could not be interrupted by EPT page faults [30]. When the PEBS buffer's memory was lazily allocated or swapped to disk, and a PEBS sample was generated, the system would crash, causing catastrophic failures that could bring down all virtual machines running on the same physical server. Although Intel mentioned a workaround [26], it required fully populating all memory pages before launching a VM and completely disabling swapping [30], which was not economically feasible for cloud vendors.

With the introduction of EPT-friendly PEBS in version 5 [48], PEBS sample writes can continue after being interrupted by an EPT page fault. Hypervisors no longer need to pre-allocate all memory pages or disable swapping, making PEBS a free bonus feature to enable in the cloud. This advancement allows guest operating systems to leverage instruction sampling efficiently and securely, facilitating low-overhead hotness tracking necessary for effective tiered memory management in virtualized environments.

Table 1: Comparison among state-of-the-art hypervisor-based tiered memory management systems

Design	HeteroOS [28]	HeteroVisor [19]	RAMinate [22]	vTMM [40]	Memstrata [51]	HyperTier
Guest Awareness	✓	✗	✗	✗	✗	✓
Guest Mgmt. Ability	✗	✗	✗	✗	✗	✓
Hotness Mgmt.	✓	✓	✓	✓	✗	✓
Locality Visibility	✗	✗	✗	✗	✗	✓
Security & Isolation	✗	✗	✗	✗	✓	✓
Memory Elaisticity	✗	✗	✗	✗	✗	✓

2.2 Hypervisor-Based TM Management Systems

Existing hypervisor-based tiered memory management systems commonly operate under the assumption that guest operating systems are incapable of managing memory tiers [19, 22, 28, 40, 51]. HeteroVisor [19] was among the first to introduce tiered memory in virtualized environments by transparently presenting a mixture of fast and slow memory to the guest OS. It introduces an *E-state* concept, inspired by the *P-state* used during CPU frequency scaling, allowing guests to request higher-performance memory mixtures by specifying different *E-state* values to the hypervisor. The hypervisor then performs memory sampling and classification to identify performance-critical pages and places them in the extra fast memory assigned.

Building on HeteroVisor, HeteroOS [28] introduced tiered memory awareness to the guest OS, enabling it to recognize different memory tiers. Despite this advancement, HeteroOS assumes that the guest OS lacks sufficient hardware privileges to perform hotness tracking, keeping tiered memory management within the hypervisor's domain.

Recent works like vTMM [40] and Memstrata [51] shift from Xen-based [8] para-virtualization to KVM-based hardware virtualization first adopted by RAMinate [22], yet they still choose hypervisor-only tiered memory management schemes. RAMinate and vTMM operate in a fully guest-transparent manner, where the guest OS is unaware of the underlying memory media and its utilization. The hypervisor in RAMinate and vTMM handles the entire hotness management process, including access sampling, hotness classification, and data migration. Conversely, Memstrata opts out of software-based tiered memory management entirely due to high hotness tracking overhead and low classification accuracy. The hypervisor's role is limited to adjusting the allocation of fast and slow memory pages to each VM based on heuristics without engaging in hotness tracking. Both approaches maintain the assumption that the guest OS cannot manage memory tiers, leaving tiered memory management exclusively to the hypervisor.

These hypervisor-centric approaches, while functional, overlook the evolving capabilities of modern guest operating systems to manage memory tiers effectively.

3 Motivation and Challenges

3.1 Responsibiliy Disaggregation

The additional layer of address translation introduced by virtualization complicates secure and isolated tiered memory management in the hypervisor. This address translation hides guest memory from the hypervisor. However, the access sampling design in HeteroVisor and HeteroOS relies on accessing the guest's PTE.A bits located in guest memory. HeteroVisor and HeteroOS use a para-virtualization architecture [8] where every guest page table modification triggers a page fault, handing control back to the hypervisor. They then mirror the guest page tables and keep them updated through these page faults. While this approach allows them to observe guests' PTE.A bits, it introduces significant overhead due to frequent context switches and page faults.

Moreover, cloud vendors have deprecated the costly para-virtualization in favor of hardware virtualization technologies [3, 6]. Mirroring guest page tables is not supported in hardware-virtualized environments. Systems like vTMM resort to monitoring the guest page table contents. Specifically, vTMM designs a guest module to capture all guest page table pages and pass their addresses in hypervisor-usable Guest Physical Address (GPA) to vTMM. vTMM then scans these guest page tables to reconstruct the hotness of guest memory based on access bits. Despite attempts to narrow down the pages to scan using Page Modification Logging (PML) [10] in vTMM, the scanning process cannot escape from the needs to access arbitrary guest memory. RAMinate tried to only rely on hypervisor managed EPT for PTE.A bits scanning, but its page remapping process still require referring to arbitrary guest memory pages.

These severe security and isolation risks present in existing hypervisor-based designs, combined with the recent advances in OS-level tiered memory support, motivate us to adopt a novel perspective in hypervisor tiered memory design: **disaggregating data placement responsibilities from tiered memory provisioning (Challenge 1).** This design philosophy allows sensitive hotness management to be carried out within the guests while retaining the elasticity of the virtualized environment. However, it also presents challenges in **how to provision tiered memory to guest virtual machines (Challenge 2)** in the first place.

The first intuitive solution would be to expose fast and slow memory as different fake NUMA nodes through ACPI table at VM boot following HeteroOS [28]. However, such assignment is static, dedicating the memory for one particular VM during its entire lifetime. Although page-

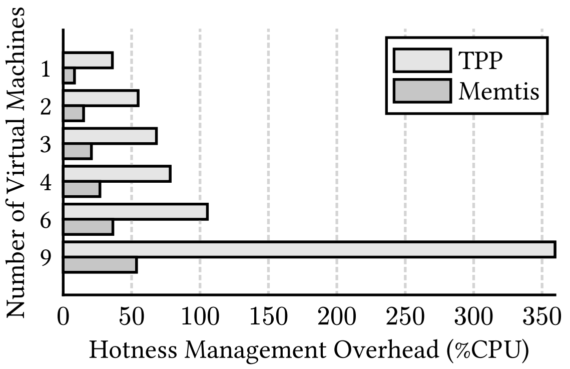


Figure 1: Sum of hotness management overhead spent when completing the same total number of GUPS transactions using different number of virtual machines.

based remapping is available for the hypervisor, it falls back into the hypervisor management trap. What's more, such remapping often require a complete stop of guest VMs during the entirety of the remapping process [22] hindering performance. Alternative soltions including provision tiered memory through virtio-mem [21] and combining the OS's ability of memory hotplugging [20]. Such solution provisions memory in blocks, allowing dynamically resizing by online and offline memory in a block granularity. However, current Linux only supports a 128MiB hotplug granularity for x86-64 architecture or even 1GiB for aarch64 [21], which is not able to satisfy the cloud's elasticity need. **These problems again post challenges of efficiency and fine-granularity on the provision solution (Challenge 3).**

3.2 Scalable Guest Management

We first experiment with existing OS-level tiered memory management systems within guest virtual machines, using the intuitive static provision solution to test the suitability of them serving the guest management responsibilities. We found that their management overhead scales with the number of virtual machines, sometimes even exponentially, preventing direct adoption.

Evaluation Platform. Our testbed is a dual-socket machine equipped with 1× 3rd Gen Intel® Xeon® Scalable 8360Y 36 Core Engineering Sample CPU at 3.0GHz, 4× 32GiB Samsung® 2R×4 DDR4 PC4-3200 ECC RDIMM, and 4× 128GiB Intel® Optane™ PMem 200 at 3200 MT/s for each socket. We run Clear Linux OS version 42660 with a modified Linux kernel version 6.10.0. Our hypervisor is a modified Cloud Hypervisor [34] based on version v36.0.0 with hardware-virtualization support from Linux v6.10.0's KVM module.

Workload. To test the feasibility of offloading tiered memory management to existing OS-level designs running under virtualization, we use the standard evaluation workload, GUPS from previous works [38, 44]. We conduct six runs, with up to nine virtual machines in each run. In each run, 36 vCPUs and 8.1 billion GUPS transactions with a 126 GiB working set are evenly divided across all virtual

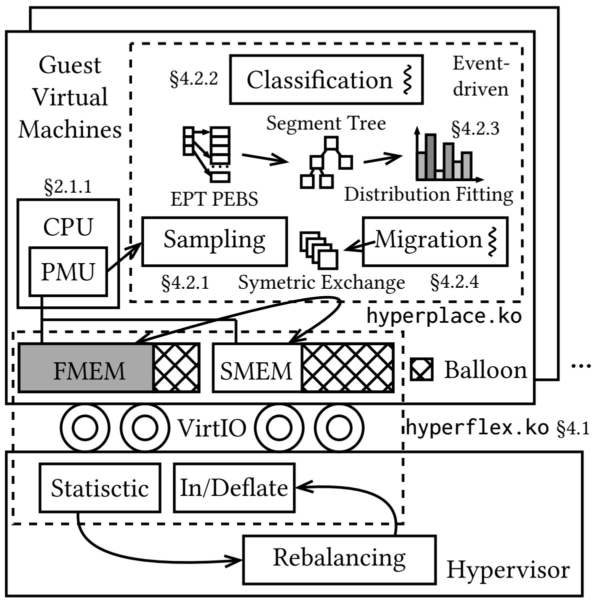


Figure 2: Design of HyperTier

machines while preserving the access distribution. We use a 4 GiB memory per vCPU configuration, with 20% allocated from DRAM serving the FMEM tier and the remainder from PMEM as the SMEM tier. We expose this two-tiered memory configuration to guests via fake NUMA nodes through the ACPI table. We measure the total management overhead and elapsed time in nanoseconds and calculate the CPU percentage. The results are shown in Figure 1.

Existing OS-level tiered memory management systems either do not limit their CPU usage [35, 47] or associate it with a certain percentage of CPU [32]. While such solutions work well on bare-metal machines, directly applying them to a multi-VM virtualized environment duplicates management overhead. As shown in Figure 1, naively using TPP under virtualization could waste more than three and a half CPU cores in a 36-core system. Cloud environments rent CPU resources to customers and often aim for maximal resource utilization. Such wastage might introduce a 10% increase in TCO, negating the benefits of memory expansion through slow memory. This motivates us to **further investigate better OS-level tiered memory management designs at every stage to minimize management overhead and maximize resource utilization.**

4 Design

As shown in Figure 2, to address the first and second challenges outlined in Section 3.1, we introduce the HyperFlex design for tiered memory provisioning. This mechanism offloads sensitive hotness management entirely to guest operating systems while ensuring fine-grained resizing, elastic overprovisioning or overcommitment, and expandibility for data center-wide scheduling. To tackle the third challenge from Section 3.2, we propose a standalone HyperPlace module for the guest OS, which uses EPT-friendly instruction sampling to collect access samples. It classifies hotness using a segment-tree structure and then places hot and cold data into fast memory and slow memory respectively after distribution fitting, using symmetric exchange. All three steps are designed to minimize overhead, ensure scalability and resource utilization We will discuss the detailed designs of HyperFlex and HyperPlace in Section 4.1 and Section 4.2.

4.1 HyperFlex: Tiered Memory Provisioning

Delegating data placement responsibilities to guest operating systems leaves the hypervisor with the challenge of effective memory provisioning. We address this challenge by introducing HyperFlex. HyperFlex first determines how to expose different memory tiers to guests with no guest modifications using the NUMA node interface, and then ensures elasticity in memory allocation for every tier using page-granular per-tier sub-balloon inflation and deflation.

HyperFlex is presented as a virtual memory balloon device emulated by the hypervisor. Guest operating systems need to only load the HyperFlex driver module to recognize this device. HyperFlex exposes hypervisor-assigned portions of FMEM and SMEM as two distinct NUMA nodes with different latency parameters. This configuration ensures the guest OS accurately perceives the memory tier topology, allowing it to manage data placement among tiers using its own mechanisms.

The HyperFlex device consists of a pair of sub-balloons, each managing one of the memory tiers. To repurpose any tier of memory for use by the host or to assign it to other virtual machines, the host inflates the corresponding sub-balloon, reclaiming memory from the guest. Conversely, to increase the provision of a memory tier to the guest OS, the host deflates the corresponding sub-balloon, returning memory to the guest. Both inflation and deflation occur at page granularity, enabling precise control over memory provisioning. The inflation process is realized by first identifying the NUMA node for the target memory tier and then allocating memory pages specifically from that node. The allocated pages are then isolated from Linux's physical page management and inserted to HyperFlex's inflated page list.

We optimize the inflation and deflation processes by making them asynchronous reducing total CPU cycles spent on resizing. The hypervisor-emulated HyperFlex device communicates with the guest HyperFlex driver using VirtIO [39]. When an inflation or a deflation request is signaled by a VirtIO interrupt, the guest HyperFlex driver's request handler executes asynchronously on the Linux kernel's `workqueue` executor. It then posts the addresses of the inflated or deflated pages to VirtIO queues dedicated to each sub-balloon. The hypervisor listens to these queues using `epoll()` on their associated event file descriptors `event-fd`. Upon receiving the page addresses, the hypervisor removes or restores the physical memory backing for these pages. Because the inflated pages are reserved from free memory, even though the hypervisor has the ability to access their content, no sensitive information is exposed, thus maintaining isolation. Finally, the hypervisor signals the completion of each request by posting an acknowledgment on the VirtIO queues.

To meet cloud elasticity requirements, guest memory can be both overcommitted and overprovisioned. Initially, each Non-Uniform Memory Access (NUMA) node of a virtual machine is assigned the full amount of required memory, and the size limit of each sub-balloon is set to the total memory required by the virtual machine VM instead of corresponding memory tier. This configuration allows memory composition to range from 100% FMEM to 100% SMEM. Moreover, the combined sizes of the two sub-balloons can range from 0% to 200% of the total required allocation.

In addition to controlling the provisioned FMEM and SMEM through inflation and deflation, we additionally expose guest memory statistics to the hypervisor via a statistics queue. These statistics enable machine- or cluster-wide memory schedulers to rebalance FMEM and SMEM across different virtual machines, ensuring quality of service.

4.2 HyperPlace: Guest OS Hotness Management

We implement our HyperPlace using PMU sampling and architect the management in a fully asynchronous, event-driven manner, where the framework of our design is laid out in Figure 2. The advantage lies in its low overhead, especially compared to traditional Linux derivatives, which can experience exponentially increasing overhead, as shown in Figure 1. HyperPlace comprises a PMU sampling overflow handler, a segment-tree-based range classification thread, and a migration thread performs symmetric exchanges.

The sampling handler ensures isolation, removes the need for polling and only requires minimal sample frequency. We minimize hotness management overhead from the origin of sample generation, using a fixed minimal event frequency and optimize the collection process. Instead of traditional busy or periodic polling when collecting PMU samples, as used in prior works [32, 38, 44], we introduce a lockless Multiple Producer Single Consumer (MPSC) queue and leverage the synchronization-by-communication principle in the sample handler. When the sample handler is either passively triggered by a Performance Monitoring Interrupt (PMI) or voluntarily executed during context switches for agility, each generated PEBS sample is passed to the classification thread via the sample MPSC queue. The classification thread is then woken up to update hotness information according to the received samples.

The classification thread can also be woken up by a classification timer. In this scenario, it ranks memory hotness based on a range-based hotness classification algorithm from Section 4.2.2. The hottest ranges in the distribution are fitted into FMEM. The classification thread sends migration requests to the migration thread via a migration MPSC queue. Migration is then carried out using symmetric exchange as described in Section 4.2.4.

4.2.1 Sampling via EPT-friendly PEBS

As introduced in Section 4.2.1, EPT-friendly, performant, and private PEBS is available to guest VMs without sacrificing memory elasticity. Our system enables the guest PEBS feature for each guest OS. HyperPlace leverages PEBS to collect memory access samples in a steady stream.

Traditional PEBS sampling designs optimize for maximum collection speed under given CPU overhead constraints by varying the sample frequency [32]. We found that this approach suffers from two drawbacks: overshoots and biased samples. PEBS samples are not directly visible right after generation; rather, software sees samples during passive draining triggered by a PMI or proactive sample draining. A PMI occurs when the PEBS buffer fills up to a configured threshold, and proactive draining happens routinely during task scheduling.

When increasing the sample frequency, the sample buffer is more likely to fill up before routine draining, triggering expensive PMIs. Because CPU resources are spent more on handling PMIs, the frequency is tuned lower. This creates a cycle where a lower sample frequency cannot utilize the allocated CPU budget, leading to tuning the frequency up again and resulting in an uneven sample stream. This inconsistency biases the sampling towards memory accessed during high-frequency periods, lowering accuracy.

Thus, we choose a small and constant sample frequency to avoid overshoots and obtain a steady stream of access samples. We find a frequency of $\frac{1}{4093}$ works well in practice, and we show that our design tolerates a wide range of sample frequencies in Figure 8. Our sample handler is invoked during context switch, draining samples from the sample buffer and sending them to the hotness classifier right away, further reduce the possibility of triggering an expensive PMI. We then carefully design our hotness classification algorithms to achieve high accuracy with a minimal number of samples, as described in Section 4.2.2. We use two events, `MEM_TRANS_RETIRED.LOAD_LATENCY` and `MEM_INST_RETIRED.ALL_STORES`, to record load and store accesses respectively.

4.2.2 Range-based Hotness Management

We leverage the insight that access locality primarily exists in an application's virtual address space [11]. Instead of using the page-centric hotness management found in existing OS-level and hypervisor-level designs, we propose a top-down management scheme based on a segment tree-like structure that divides and identifies virtual addresses into ranges of interest, as illustrated in Figure 3.

We use range-based approach to have a gradient on classification resolution, minimizing tracking overhead on cold memory and maximize tracking accuracy on hot data by splitting the more frequently accessed ranges into expoentially smaller sub-ranges agilely.

Range split. We discover the hottest ranges through a series of split operations. After collecting an epoch's

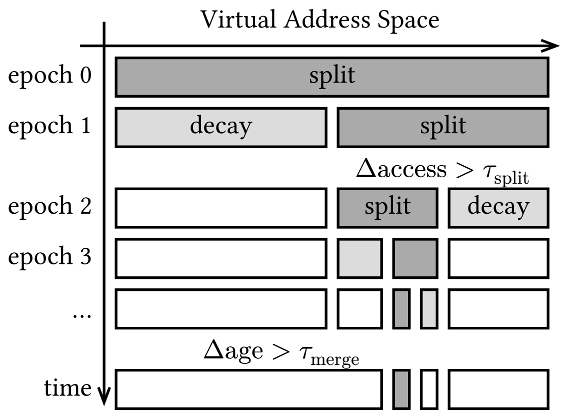


Figure 3: Design of the range-based classification algorithm using on a segment tree like structure.

worth of samples, we check whether any ranges need to be split. We examine all the leaf-level ranges to determine if any have significantly more accesses than both of their neighbors. Since memory accesses can come from all CPU cores, we define a significance factor $\alpha = \frac{\Delta\text{access}}{\tau_{\text{split}} \cdot \text{vcpu}}$, where Δaccess is the difference in memory accesses compared to a neighbor, τ_{split} is the split threshold and vcpu is the total number of virtual CPUs. When a split happens, we will split the range in the middle into two halves with equal sizes. We choose an epoch of 500 ms, α of 2, and τ_{split} of 15. When a split occurs, we divide the range into two equal halves. The access count for each new range is set to half the total access count of the original range. We do not split ranges beyond the split granularity, currently set to 2 MiB. Each range also has two `age` fields that track when it was created during the split operations and when its accesses were last found. During each epoch, the access count for each region is halved to ensure older accesses gradually decay to zero.

Hotness classification. After the splitting process, we calculate each range's hotness frequency by dividing its total access count by its size. We rank all the ranges based on these hotness frequencies. If frequencies are equal, we use the range's creation age as a tiebreaker, leveraging temporal locality, that newly created ranges might be accessed more frequently in the near future. We aim to fit as many hot ranges as possible into FMEM based on current capacity and the access distribution calculated earlier. We will discuss the detailed migration design in Section 4.2.4.

Our hotness classification process is agile. For a modern process with a 48-bit virtual address space, excluding kernel space addresses, which have the most significant bit set, we have a 128 TiB application virtual address space. After excluding memory segments that always need to be in fast memory, we are left with approximately a 40 TiB range to manage. With a 500 ms epoch length, finding the smallest hot spot of a 2 MiB range requires about 24 split operations within 12 seconds, creating fewer than 50 ranges, which takes negligible time to manage or rank. The application's virtual address space is sparse, resulting in only a few deep branches of small ranges in the tree, while the rest are large ranges with infrequent accesses. The total number of ranges is expected not to exceed several hundred. Ranges can also be merged to further reduce the total number of ranges to manage. If neighboring ranges' access counts have decayed to zero and there have been τ_{merge} splits since then, they are merged into a single range.

4.2.3 Distribution Fitting

After ranges are ranked by hotness during each epoch, we try to find the largest index `f` such that the total number of intersecting pages in ranges with indices in `[0, f)` is less than the total fast memory capacity. These ranges are our promotion candidates. We traverse the process's page table intersecting these ranges to identify pages not yet promoted and collect them into a promotion linked list. The page table is walked linearly in a cache friendly manner, rather than reversely through the `rmap` structure as required in PTE.A bits collection. For the range at the boundary index `f`, we walk the first `n` pages, where `n` is the difference between the total fast memory capacity and the total number of pages intersecting ranges in indices `[0, f)`, aiming to fit as many pages into fast memory as possible.

4.2.4 Symetric Exchange

After the collection of a promotion linked list in Section 4.2.3, we then form a demotion linked list with a matching number of pages by traversing the coldest ranges and finding their intersected pages that are in fast memory. We send the exchange request containing these two lists to the migration thread asynchronously via migration `mpsc`.

The migration thread performs symmetric exchange rather than paired migration. During a symmetric exchange, the promotion and demotion candidates are unmapped from their page tables, data are swapped, and finally remapped. No extra memory allocation is needed, and there is no fluctuation in free memory watermarks that might wake up `kswapd` or trigger on-critical-path direct reclamation to free up fast memory, causing unwanted demotion. Unlike previous designs where promotion happens after demotion and requires enough memory headroom reclaimed by rotating the LRU and collecting PTE.A bits, our symmetric exchange avoids these issues.

Our symmetric exchange should not be confused with page exchange mentioned in previous works [29, 40, 51], which is asymmetric and performs paired migration.

4.3 Limitations

Hugepage. Considering the hugepage unalignment issue across guests and hypervisors reported by previous work [27], we did not integrate huge page support in our design. However, our sample collection and hotness identificaiton design is agnostic to hugepages and work well for any page granularity. It is also intuitive to add hugepage support to our symetric exchange by adding two additional linked lists for matching promoting and demoting hugepage.

As for HyperFlex, we extend the VirtIO balloon specification [42] with no support for hugepages. To add hugepage support, we could introduce an additional feature bit, allowing the negotiation of page sizes among hypervisor and guests. The inflation and deflation process can then follow the negotiated page sizes to reserve or free up memory.

QoS. Although we have the infrastructure for rebalancing tiered memory allocation across guests in place. Quality of service scheduling is a delicated dance and require detailed tailoring. We leave machine- or cluster-wide QoS control for the future work.

5 Implementation

HyperFlex. We implement HyperFlex by extending the traditional memory balloon device described in the VirtIO [39] specification [42], repurposing the traditional configuration space and virt-queues for inflation and deflation to control the FMEM sub-balloon. SMEM sub-balloon is then implemented thourgh mirroring the FMEM sub-balloon's control structures and virt-queues appending to the original configuration space and virt-queue list respectively. The HyperFlex extension is guarded behind an additional tiered memory feature flag, maintaining compability with traiditional VirtIO memory balloon drivers. The HyperFlex device and driver are implemented via 1039 lines of Rust code and 839 lines of C code respectively as part of Cloud Hypervisor [34] and Linux Kernel.

HyperPlace. HyperPlace is implemented as a standalone Linux kernel module, allowing easily integreting to various kernel versions with minimal modifications. HyperPlace module is implemented in an additional 8257 lines of C code.

Range Initialization. We focus on the process with the largest memory footprint, avoiding overhead for those with minimal memory usage. We also provide a `sysfs` interface allowing managing multiple target processes in parallel. When a target is selected, we create a hotness tree root node covering `[start_brk, mmap_base)`. We exclude frequently accessed program text, static data, and the stack, ensuring they remain in FMEM. Since allocations occur through `brk()` (growing upward) and `mmap()` (growing downward), all memory lies within the heap and mmap segments bounded by `start_brk` and `mmap_base`.

6 Evaluation

We first demonstrate how HyperFlex enables guest operating systems to manage hotness through elastic, efficient tiered memory provisioning. Next, we measure the performance, efficiency, and scalability of HyperPlace using the GUPS workload and then test our full system on real-world applications including databases, scientific computing, graph processing, and machine learning. Our experiments use two tiered memory configurations: DRAM combined with either real PMEM or emulated CXL.mem. We compare HyperTier's HyperFlex against static provisioning (VirtIO-Mem [21]) and elastic provisioning (VirtIO-Balloon [39]), as well as evaluate HyperPlace against PMU sampling–based solutions like Memtis [32] and Linux derivatives such as Nomad [47], TPP [35]. We cannot reproduce or compare with hypervisor-based designs due to the unavailability of their source code and the lack of implementation details.

Table 2: Memory access latency and bandwidth matrix measured by the Intel Memory Latency Checker [25]

Access to	L2	L-DRAM	R-DRAM	L-PMEM
Latency (ns)	53.6	68.7	121.9	176.6
Bandwidth (MB/s)	-	88156.5	53533.8	21414.5

Our evaluation addresses three key questions: ① Can HyperFlex transform Linux derivatives [35, 47] and specialized tiered memory OSs [32] into virtualization-ready solutions (Section 6.2)? ② Can HyperPlace improve performance while minimizing overhead (Section 6.3)? ③ Are HyperPlace's throughput and runtime gains consistent across different configurations and complex workloads (Section 6.4 and Section 6.5)?

6.1 Experimental Setup

Hardware Setup. We conducted all evaluations on the same dual-socket machine described in Section 3.2. Each socket contains one CPU node and two locally attached memory nodes: one DRAM node and one PMEM node. All vCPUs for a single virtual machine are allocated from the same CPU node to avoid NUMA effects from cross-node allocations, which are beyond the scope of this work.

Although our machine supports hyperthreading, providing 72 threads per CPU node, we use only 36 threads mapped to separate physical cores to avoid performance variability caused by hyperthreading. We allocate up to 36 vCPUs to virtual machines, all from the same CPU node. We lock the CPU frequency to the maximum achievable value under full load (3.0 GHz). The memory access performance of our testbed is shown in Table 2.

Unless specified otherwise, each virtual machine is configured with 4 vCPUs and 16 GiB of tiered memory, following a common configuration in public clouds [13]. Tiered memory is managed via the NUMA interface on the host.

We explore two types of tiered memory composition:

1. **DRAM with real PMEM.** We allocate fast memory from locally attached DRAM (L-DRAM) and slow memory from PMEM attached to the same CPU package (L-PMEM). PMEM is exposed to the host via `devdax` mode. We create a PMEM NUMA node backed by the `devdax` device using the `daxctl` utility.
2. **DRAM with emulated CXL.mem.** Since we do not have access to real CXL.mem expansion cards, we emulate CXL.mem using remote DRAM (R-DRAM). This configuration exhibits similar performance to real

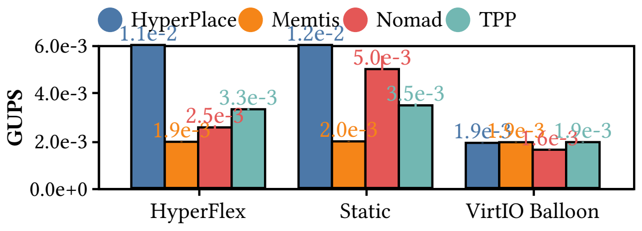


Figure 4: GUPS throughput comparison among tiered memory provision designs (higher the better)

CXL.mem devices attached via PCIe without switching overhead [33].

The overall performance under both settings is similar. However, due to the limited capacity of R-DRAM (128 GiB) compared to L-PMEM (512 GiB), we primarily present results obtained under configuration 1, which allows for higher scalability. The default fast memory ratio for each virtual machine is set to $\frac{1}{5}$, reflecting the physical DRAM to PMEM capacity ratio. We discuss the performance of configuration 2 in Section 6.5.

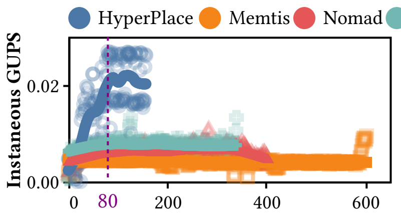
6.2 Tiered Memory Provision

Workload. We use the GUPS workload [38] as our microbenchmark to evaluate memory performance. GUPS measures how many billion read-modify-write transactions can be performed per second. We utilize the `hotset` variant of GUPS, which divides the memory space into a hot section and a cold section. The hot section is accessed ten times more frequently than the cold section, with random memory accesses within each section.

Each worker thread, one per CPU, carries out these transactions. The original GUPS implementation assigns transactions statically to each worker thread[[38], which can lead to idle workers near the end of a run, causing inaccurate measurements. To address this, we use a modified GUPS implementation with work-stealing transaction assignments. In this setup, a worker thread that finishes early will steal transactions from threads with longer queues, ensuring balanced utilization.

When not specified otherwise, we configure GUPS with a working set occupying $\frac{7}{8}$ of the total memory available in the virtual machine. The hotset constitutes $\frac{1}{7}$ of the total working set. In each run, we perform three iterations of 900 million transactions each. The first two iterations serve as warm-up, and we measure performance during the last iteration.

Provision Benchmark. We run nine virtual machines simultaneously, provisioning tiered memory through three methods: static allocation using VirtIO-Mem, elastic provisioning using VirtIO-Balloon, and our HyperFlex. We present the average GUPS throughput and standard error across all virtual machines in Figure 4. For each provisioning method, we also report results using different guest memory management solutions.



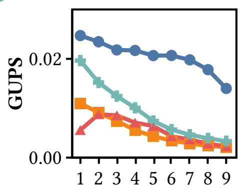


Figure 5: Instantaneous GUPS Figure 6: Scalability

Static tiered memory allocation yields the best performance across all guest hotness management designs. However, static provisioning lacks elasticity and relies on coarse-grained memory hot-plugging for online VM resizing, which contradicts cloud requirements discussed in Section 3.1. For elastic provisioning, we allocate twice the required memory and rely on VirtIO-Balloon or HyperFlex to resize to the desired memory profile dynamically.

VirtIO-Balloon allows page-granular online resizing but is unaware of different memory tiers. Its inflation conflicts with the Linux kernel's allocator design, causing the system to prioritize reclaiming fast memory, which leads to underprovisioning of fast memory. Even with guest tiered memory management enabled, all four designs have insufficient fast memory to place hot data, resulting in similarly poor performance.

In contrast, HyperFlex is aware of memory tiers and reclaims memory according to the configured tiered memory budget and fast memory ratio. This results in allocations nearly identical to static provisioning. The performance of three of the four guest designs using HyperFlex is nearly identical to the optimal case of static provisioning, with less than a 10% slowdown for TPP and Memtis.

Notably, for the best-performing existing OS-level design, TPP, integrating it with HyperFlex to create a virtualization-ready tiered memory design yields a 74% performance improvement compared to the existing elastic provisioning design, VirtIO-Balloon. These results demonstrate HyperFlex's ability to transform an OS-level tiered memory solution into an elastic, virtualization-ready design.

6.3 Efficient and Scalable Data Placement

Agile Hotset Classificaiton. In this test, we use the same GUPS workload but measure the instantaneous GUPS from the very start of the benchmark without any warm-up period. We plot the per-second instantaneous GUPS in Figure 5 as points and display the running average using a locally estimated smoothing curve. This benchmark evaluates how quickly different systems can identify the hot set.

Initially, all systems show low instantaneous GUPS because the hot set is located in slow memory and has not yet been identified. Over time, we observe an increasing trend in GUPS for HyperPlace, Nomad, and TPP, indicating that these designs begin to recognize the hot set. However, Memtis shows no significant change in instantaneous GUPS, suggesting it fails to detect the hot set. HyperPlace

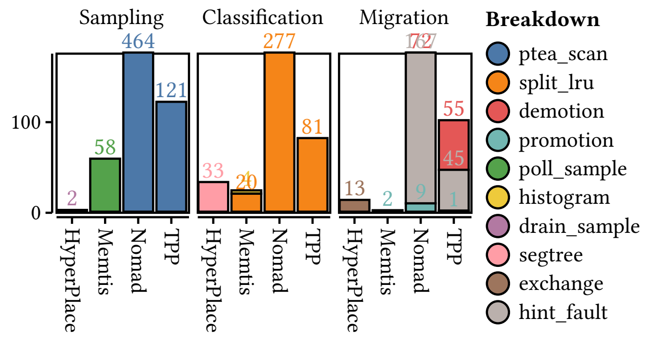


Figure 7: Breakdown of average overhead spent in seconds when running nine GUPS workload (lower the better)

achieves the highest instantaneous GUPS in less than 80 seconds, demonstrating its ability to detect a larger portion of the hot set more quickly than other designs and efficiently move the hot data to fast memory. While Nomad and TPP also increase their instantaneous GUPS early in the benchmark, they do not identify as much of the hot set as HyperPlace and maintain lower GUPS throughout the run.

Scalability. To assess how well HyperPlace performs in a multi-tenant cloud environment, we run between one and nine virtual machines in parallel, each executing the same GUPS hotset workload, and record their performance to evaluate scalability. We are limited to nine virtual machines due to the number of available cores. We plot the average GUPS throughput and standard error in Figure 6. HyperPlace maintains relatively stable performance with up to seven virtual machines. In contrast, the second-best system, TPP, experiences a steady decline in performance as the number of co-running virtual machines increases, eventually converging with the other systems when more than five virtual machines are running.

Overhead Breakdown. We further analyze the overhead in all three stages of tiered memory management: access sampling, hotness classification, and page migration. We run nine virtual machines in parallel with GUPS workloads. We measure the CPU time spent in each stage using the `vmstat` interface with nanosecond precision, aggregate the average results, and present them in Figure 7.

Although we use PEBS as Memtis does, our access sampling overhead is only $\frac{1}{29}$ of that in Memtis. While Memtis has lower classification and migration costs compared to our design, it fails to correctly detect the hot set in the GUPS workload, as shown in Figure 6. Our segment-tree-based range classification algorithm and symmetric page exchange outperform the split-LRU and paired migration used in Nomad by up to 8.4× and 18.4×, respectively, and also outperform TPP by 2.5× and 7.7×, respectively.

6.4 Parameter Sensitivity Study

We evaluate how varying parameters affects our sampling and classification designs by running five virtual machines with the GUPS workload in parallel. Figure 8 shows the

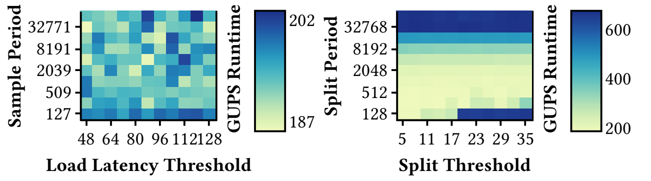


Figure 8: GUPS runtime under different sampling and classification parameter settings (lighter the better)

results. Our sampling design records memory access samples when latency exceeds a threshold, collected at intervals defined by the sample period. GUPS performance remains stable, varying by less than 10% under different settings. The best results appear with a threshold near 80 and a period around 4093. For hotness classification, we split the hottest ranges based on a split period when per-CPU access counts exceed a threshold. GUPS continues to perform well, completing in about 200 seconds. The optimal configuration occurs with a period near 512 and a threshold of 11.

6.5 Real-world Workloads

Workloads. Following Memtis [32], we select seven representative memory-intensive real-world workloads from databases, scientific computing, graph processing, and machine learning. For databases, we choose a B-tree database index workload from Mitosis [2] and an in-memory online transactional database engine, Silo [43]. For scientific computing, we select bwaves (603.bwaves_s), a blast wave simulator for fluid dynamics from SPEC CPU 2017 [12], and XSBench [41], a nuclear reactor particle simulator. We choose bwaves due to its scalability and adaptability to different memory sizes. For graph processing, we include graph500 [1], which uses generated Kronecker graphs, and PageRank [9] applied to a real-world social network graph from Twitter [31]. Finally, for machine learning, we select multi-core LibLinear [17], a large-scale linear classification workload with the `kdda` dataset. We scale up each workload as much as possible without exceeding our test virtual machine's memory capacity.

Performance. We present the results in Figure 9. We use the elapsed wall-clock time as the performance metric across all workloads and run up to nine virtual machines in parallel. For each subgroup of results running the same workload with the same number of virtual machines, we also show the relative elapsed time ratios among all designs. Our design performs best in five out of the seven workloads, achieving speeds 1.1× to 9.7× as fast. Compared to the second-best design, we improve performance by up to 90%. Although we did not outperform all other designs in two of the graph processing workloads, we are the second-best alternative in those cases, with less than a 10% slowdown. Our design excels in workloads with relatively skewed access patterns because we can promptly identify and react to hot data regions, as demonstrated in Section 6.3. For example, in LibLinear, the model weights are updated itera-

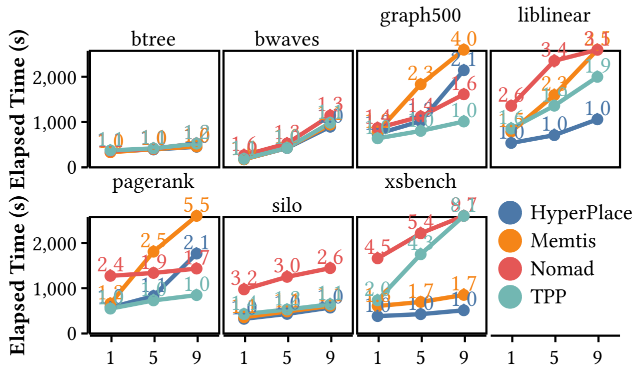


Figure 9: Average per-VM elapsed time under different number of parallel running virtual machines and different real-world workloads on real PMEM (lower the better)

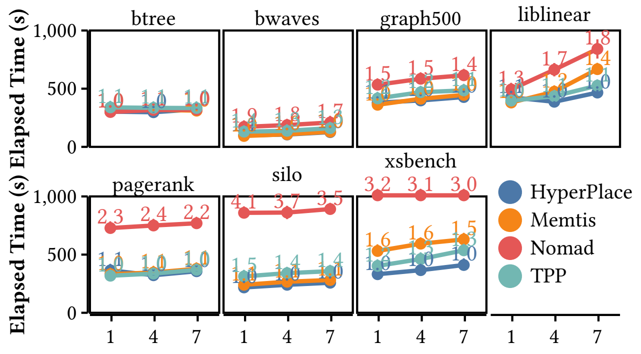


Figure 10: Real-world Workload on emulated CXL.mem

tively, while the labeled data are accessed less frequently, resulting in a hotspot that our design captures. In contrast, the B-tree workload performs random index lookups to leaf nodes with nearly no hotspot. Although B-tree traversal begins from the root node and its immediate children, these common paths are too small to form a significant hotspot, leading to similar performance among all designs. However, our design still ranks among the top performers due to better management overhead control.

Emulated CXL.mem. We run the same set of real workloads as Figure 9, but change the slow tier from real PMEM to emulated CXL.mem and plot the results in Figure 10. Because of the reduced capacity of emulated CXL.mem compared to PMEM in our testbed, we can only run upto seven virtual machines in parallel before triggering an out-of-memory kill and interrupting the benchmark.

Our design achieves upto 10%, 90%, 50%, 80%, 140%, 310% and 220% performance improvements across `btree`, `bwaves`, `graph500`, `liblinear`, `pagerank`, `silo` and `xsbench` respectively. And compared to the overall second best design TPP, our design achieves upto 50% improvement on `silo`.

7 Discussion

Hardware tiering and caching are currently supported only by Intel CPUs through Flat Memory Mode (FMM) [51] and PMEM's Memory Mode (PMM) [50], respectively. PMM treats all available DRAM as a transparent, inclusive, direct-mapped cache for PMEM. Enabling PMM sacrifices all DRAM capacity, slows down every memory access, and eliminates software control over placement strategies and quality-of-service management. In contrast, FMM combines a portion of DRAM with CXL.mem to create a slower memory tier with combined capacity. However, FMM trades off memory bandwidth due to frequent, transparent swapping of data between CXL.mem and DRAM in the memory controller when the accessed address is not in DRAM no matter the hotness. Moreover, the performance of this hardware tier remains inferior to that of pure DRAM and still requires dedicating DRAM to boost application performance [51].

Sampling using PML. Another hardware-accelerated access profiling technology is Page Modification Logging (PML) [40], which is similar to PMU sampling but performs worse in terms of overhead, isolation, and versatility. PML logs memory access addresses into a small, fixed-size buffer that can store up to 512 records. Unlike PMU sampling, which supports sampling modes, PML logs every access, generating a large number of interrupts to synchronously drain these records when the buffer is full. The samples generated from different virtual machines are directly visible to the hypervisor, violating isolation and lacking virtualization support. Furthermore, unlike PMU, PML hardware is not widely available across different architectures.

Optimizing for access latency instead of hot data coverage [44] is an alternative perspective proposed by Collid. However, this method inherently conflicts with virtualization. It relies on a shared state observed by multicore shared hardware, which breaches isolation. Additionally, when one virtual machine adjusts its data placement to reduce load on a memory tier, others may make opposite decisions, increasing the load. These conflicting adjustments can cascade, leading to significant contention among virtual machines and memory tiers.

8 Conclusion

HyperTier integrates tiered memory into virtualized cloud environments by transforming OS-level tiered memory management systems into elastic, virtualization-ready solutions. Through HyperFlex, we enable existing systems to adapt to virtualization, achieving a 74% performance improvement. Our optimized HyperPlace design further reduces overhead and boosts performance by up to 90% on real-world workloads compared to the next best alternative. Evaluations with configurations combining DRAM with real PMEM or emulated CXL.mem confirm the effectiveness of our approach. HyperTier addresses isolation, elasticity, and efficiency challenges, offering scalable and cost-effective tiered memory solutions for the cloud.

References

[1] Graph 500 Benchmarks. Retrieved from https://graph500.org/

[2] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020. Association for Computing Machinery, New York, NY, USA, 283–300. https://doi.org/10.1145/3373376.3378468

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, February 2020. USENIX Association, Santa Clara, CA, 419–434. Retrieved from https://www.usenix.org/conference/nsdi20/presentation/agache

[4] 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 '17)*, 2017. Association for Computing Machinery, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[5] Amazon. Amazon EC2 Instance types. Retrieved from https://aws.amazon.com/ec2/instance-types/

[6] Amazon. 2017. Compute-Intensive C5 Instances for Amazon EC2. Retrieved from https://aws.amazon.com/blogs/aws/now-available-compute-intensive-c5-instances-for-amazon-ec2/

[7] Andrea Arcangeli. 2012. AutoNUMA: the other approach to NUMA scheduling. Retrieved from https://lwn.net/Articles/488709/

[8] 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. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003. Association for Computing Machinery, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[9] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* (2015). Retrieved from http://arxiv.org/abs/1508.03619

[10] Stella Bitchebe, Djob Mvondo, Alain Tchana, Laurent Réveillère, and Noël De Palma. 2020. Intel Page Modification Logging, a hardware virtualization feature: study and improvement for virtual machine working set estimation. *CoRR* (2020). Retrieved from https://arxiv.org/abs/2001.09991

[11] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. 2024. A Tale of Two Paths: Toward a Hybrid Data Plane for Efficient Far-Memory Applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, July 2024. USENIX Association, Santa Clara, CA, 77–95. Retrieved from https://www.usenix.org/conference/osdi24/presentation/chen-lei

[12] Standard Performance Evaluation Corporation. 2017. SPEC CPU® 2017. Retrieved from https://www.spec.org/cpu2017/

[13] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[14] Dell. 2024. Dell Memory Upgrade - 64 GB - 4Rx4 DDR4 LRDIMM 2666 MT/s. Retrieved from https://web.archive.org/web/20241105065702/https://www.dell.com/en-us/shop/mem/apd/a9781930/

[15] Dell. 2024. Dell Memory Upgrade - 128 GB - 2666 MT/s Intel Opt DC Persistent Memory. Retrieved from https://web.archive.org/web/20241105065608/https://www.dell.com/en-us/shop/mem/apd/aa664973/

[16] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23)*, 2023. Association for Computing Machinery, New York, NY, USA, 727–741. https://doi.org/10.1145/3582016.3582031

[17] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A Library for Large Linear Classification. *J. Mach. Learn. Res.* 9, (June 2008), 1871–1874.

[18] Google. General-purpose machine family for Compute Engine. Retrieved from https://cloud.google.com/compute/docs/general-purpose-machines

[19] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '15)*, 2015. Association for Computing Machinery, New York, NY, USA, 79–92. https://doi.org/10.1145/2731186.2731191

[20] Dave Hansen. 2005. [RFC][PATCH] Memory Hotplug. Retrieved from https://lore.kernel.org/linux-mm/1108685111.6482.40.camel@localhost/

[21] David Hildenbrand and Martin Schulz. 2021. Virtio-Mem: Paravirtualized Memory Hot(Un)Plug. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '21)*, 2021. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3453933.3454010

[22] Takahiro Hirofuchi and Ryousei Takano. 2016. RAMinate: Hypervisor-based Virtualization for Hybrid Main Memory Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*, 2016. Association for Computing Machinery, New York, NY, USA, 112–125. https://doi.org/10.1145/2987550.2987570

[23] Intel. 2014. Intel® Xeon® Processor E7-8895 v2. Retrieved from https://www.intel.com/content/www/us/en/products/sku/79209/intel-xeon-processor-e78895-v2-37-5m-cache-2-80-ghz/specifications.html

[24] Intel. 2024. Intel® Xeon® 6780E Processor. Retrieved from https://www.intel.com/content/www/us/en/products/sku/240362/intel-xeon-6780e-processor-108m-cache-2-20-ghz/specifications.html

[25] Intel. 2024. Intel® Memory Latency Checker v3.11b. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[26] Intel. 2024. Intel® 64 and IA-32 Architectures Software Developer Manuals. Retrieved from https://web.archive.org/web/20240816185003/https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[27] 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 (EuroSys '23)*, 2023. Association for Computing Machinery, New York, NY, USA, 298–313. https://doi.org/10.1145/3552326.3567487

[28] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*, 2017. Association for Computing Machinery, New York, NY, USA, 521––534. https://doi.org/10.1145/3079856.3080245

[29] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, July 2021. USENIX Association, 715–728. Retrieved from https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[30] Andi Kleen. 2014. Implement PEBS virtualization for Silvermont. Retrieved from https://lore.kernel.org/all/1401412327-14810-1-git-send-email-andi@firstfloor.org/

[31] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proceedings of the 19th international conference on World wide web*, 2010. ACM, New York, NY, USA, 591–600. https://doi.org/https://doi.acm.org/10.1145/1772690.1772751

[32] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, 2023. Association for Computing Machinery, New York, NY, USA, 17–34. https://doi.org/10.1145/3600006.3613167

[33] 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 '23)*, 2023. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3575693.3578835

[34] LF Projects LLC. 2023. Cloud Hypervisor. Retrieved from https://www.cloudhypervisor.org/

[35] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23)*, 2023. Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[36] Microsoft. 2024. Sizes for virtual machines in Azure. Retrieved from https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/overview

[37] Atish Patra. 2024. RISC-V SBI v2.0 PMU improvements and Perf sampling in KVM guest. Retrieved from https://lore.kernel.org/kvm/20240416184421.3693802-1-atishp@rivosinc.com/

[38] 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 *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, 2021. Association for Computing Machinery, New York, NY, USA, 392–407. https://doi.org/10.1145/3477132.3483550

[39] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev. (OSR '08)* 42, 5 (July 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[40] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2023. vTMM: Tiered Memory Management for Virtual Machines. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*, 2023. Association for Computing Machinery, New York, NY, USA, 283–297. https://doi.org/10.1145/3552326.3587449

[41] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, 2014. Kyoto. Retrieved from https://www.mcs.anl.gov/papers/P5064-0114.pdf

[42] Michael Tsirkin and Cornelia Huck. 2023. Virtual I/O Device (VIRTIO) Version 1.3. Retrieved from https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.html

[43] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, 2013. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[44] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the 30th Symposium on Operating Systems Principles (SOSP '24)*, 2024. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3694715.3695968

[45] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI 02)*, December 2002. USENIX Association. Retrieved from https://www.usenix.org/legacy/events/osdi02/tech/waldspurger.html

[46] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 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 '22)*, 2022. Association for Computing Machinery, New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731

[47] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, July 2024. USENIX Association, Santa Clara, CA, 19–35. Retrieved from https://www.usenix.org/conference/osdi24/presentation/xiang

[48] Like Xu. 2022. perf/x86/intel: Expose EPT-friendly PEBS for SPR and future models. Retrieved from https://lore.kernel.org/kvm/20220922051929.89484-1-likexu@tencent.com/

[49] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019. Association for Computing Machinery, New York, NY, USA, 331–345. https://doi.org/10.1145/3297858.3304024

[50] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, February 2020. USENIX Association, Santa Clara, CA, 169–182. Retrieved from https://www.usenix.org/conference/fast20/presentation/yang

[51] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, July 2024. USENIX Association, Santa Clara, CA, 37–56. Retrieved from https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong

Significant Changes from Previous Version

In response to reviewer feedback about incremental novelty and design motivation, we have made the following substantial improvements:

1. **Unified Theoretical Framework**

   We have fully developed the ***guest-delegation*** principle as a cohesive theoretical framework underlying our approach, clarifying how this represents a paradigm shift rather than an incremental improvement over existing hyperivsor-based and kernel-based solutions.
2. **Strengthened Motivation**

   We have added detailed empirical analysis in Section 2.3 demonstrating the fundamental limitations of hypervisor-based approaches, including:
   - Quantitative measurement of TLB flushes and overhead (Tbl. 1)
   - Analysis of EPT-friendly PEBS scalability issues (Fig. 2)
3. **Expanded Evaluation**

   We have addressed key evaluation gaps by:
   - Adding direct comparisons with hypervisor-based approaches (TPP-H) across seven real-world workloads
   - Including latency analysis for interactive applications (Fig. 12), showing 23% reduction in tail latency
4. **Clarified Novelty of PEBS Approach**

   We have substantiated our claims about EPT-friendly PEBS with:
   - Technical explanation of previous architectural limitations (Section 2.3.2)
   - Analysis of isolation properties ensuring multi-tenant security
   - Context-switch-based sampling implementation that reduces CPU overhead by 16×
5. **Added Threat Model Discussion**

   Despite this is not the main focus of our paper, we have included a threat model in Section 7.2, addressing security considerations for untrusted guest code running in multi-tenant environments.