HNVM: Hybrid NVM Enabled Datacenter Design and Optimization

Yanqi Zhou[†] Ramnatthan Alagappan[§] Amirsaman Memaripour[*]

Anirudh Badam David Wentzlaff[†]

[†]Princeton University [§] U. of Wisc., Madison [*] UCSD Microsoft

Abstract

Emerging non-volatile memories (NVMs) like battery-backed DRAM and phase-change memory offer durability at higher speeds than traditional storage. Each technology represents a different point in the cost, performance and endurance space. In this paper, we propose that storage intensive applications use more than one kind of NVM in configurable proportions to strike a balance between cost, performance and endurance.

Furthermore, in virtualized environments, we show that a fixed ratio of different NVMs across co-located applications may not be ideal. We propose a new resource allocation and scheduling algorithm to enable cloud providers to help co-located applications use different proportions of NVM while meeting latency and capacity requirements. We evaluate the proposed design by simulating several storage intensive workloads and demonstrate average cost savings of 33% for cloud providers while degrading performance by only 4% and not violating any service-level agreements. Compared to conventional scheduling algorithms, the proposed scheduling algorithm reduces a cloud provider's penalty cost and operation cost by more than 20%, 50%, and 50% at low, medium, and high server loads.

1. Introduction

Embracing diverse non-volatile memories (NVMs) can help improve overall performance and control costs. NVMs such as battery-backed DRAM (BBRAM), phase-change memory (PCM), Intel/Micron's 3D-Xpoint, and memristors have the potential to offer data persistence at unprecedented performance. They will be added to an already diverse set of storage solutions that differ greatly in performance, cost and endurance aspects. In this paper, we show how to deal with storage diversity while meeting performance and cost constraints.

NVMs can be relatively easily incorporated into single-node designs to provide faster layers of storage, but the current architecture of cloud computing deployments imposes additional challenges. These deployments segregate computing from storage into separate services. Whenever working data needs to persist, it is flushed from virtual machines running on compute nodes into a storage service, which is typically slow due to network and software overheads. Additionally, cloud workloads commonly require high availability and fault tolerance from the infrastructure, resulting in replication of data across multiple independent nodes to protect against hardware and software failures. For these reasons, we study the deployment of NVM in datacenters as a fast and replicated cache storage layer between compute and storage services. Although, our techniques are relevant to systems that may use NVM as the storage technology.

Each NVM technology has distinct characteristics that can be used to improve cloud services. For example, BBRAM is non-volatile, has no endurance issues, and performs as well as DRAM. However, its cost and limited capacity prevent it from being used as the single storage solution in the datacenter. Another example is PCM, which is expected to be much denser and to cost significantly less than BBRAM, but suffers from endurance issues and is not as fast as BBRAM. Despite the variety of points in the space, there is no single NVM technology that is clearly superior to others.

Unfortunately, customers run diverse sets of applications with different requirements and utility functions. As such, no single NVM technology is adequate for all workloads that run in the datacenter. Furtheremore, since such diverse applications have to be colocated on the same physical nodes, no single ratio of multiple NVM technologies is right for all the applications.

We propose to provision datacenter compute nodes with a hybrid NVM compute-side cache called HNVM. To quantify the tradeoff between NVM ratios and the resulting performance and cost, we design a service-level agreement (SLA) model centered on request throughput and *latency distributions* instead of resources, giving cloud providers more flexibility in delivering the contracted service by adjusting the ratio of different NVMs.

We use BBRAM and PCM in a hybrid NVM cache case study. The cache is three-way replicated across different machines with a master and two slaves. Bare metal machines are uniform with identical ratios of BBRAM to PCM but workloads co-located on these machines are allocated different proportions of BBRAM and PCM. We study the behavior of various storage intensive workloads under different BBRAM/PCM scenarios. We develop a profiling-based model of interference between these workloads to inform the resource allocation algorithm about the impact of different workload co-locations. Using these intereference models, we propose a new scheduling algorithm to help the provider to efficiently colocate applications on shared hardware while meeting individual SLAs.

We evaluate the proposed design by simulating several stor-

age intensive cloud workloads and demonstrate average cost savings of 33% for cloud providers while degrading performance by only 4% and not violating any SLAs. Compared with conventional scheduling algorithms, the proposed CostFit scheduling algorithm reduces a cloud provider's penalty cost and operation cost by more than 20%, 50%, and 50% at low, medium, and high server loads.

2. Background and Motivation

NVM in the cloud. The volatile:non-volatile dichotomy is starkly evident in today's cloud services. For example, cloud compute services like Amazon's EC2 [2] and Azure Compute [4] are entirely stateless – vis-a-vis, any data stored locally within the VM is not durable. Applications in the cloud store state durably via separately run services such as Amazon's S3 [3], Amazon's EBS [1] and Azure Storage [5].

The separation of compute and storage services fueled their growth and helped them scale. For example, statelessness allowed cloud compute systems to deal with VM failures, to enable easy migration of VMs and also to help providers consolidate VMs as they saw fit to reduce costs. Having cloud storage isolated from these migrations helps storage services abstract internal hardware and software failures from the rest of the world and help achieve a singular important goal of never losing data and making it always available to the outside world.

This dichotomy, however, creates a challenge for NVM adoption in the cloud: Non-volatility in a compute VM is not useful if VMs do not have affinity to local resources. Low-latency cannot be exploited if NVM is added to the storage service which can be reached only via several layers of software within compute and storage services.

Cloud services must be redesigned to adopt NVM in a manner such that VMs continue to remain stateless while storage services are isolated, so that they can deal with failure internally rather then exposing them to the outside world; yet, they must expose NVM in a manner such that its low-latency persistence can be exploited by cloud applications. In this work, we analyze an architecture that layers NVM as a replicated, durable and fault-tolerant cache between the cloud compute and storage services.

NVM programming and replication. NVM programming libraries provide a memory management interface along with semantics that enable programmers to perform byte-addressable and transactional (Atomicity, Consistency, Isolation and Durability, or ACID) updates to NVM. Intel's persistent memory library, libpmem [14], is one such library, and has recently been getting traction. It provides useful APIs for applications to use persistent memory. We used a modified version of libpmem to implement HNVM.

The modifications were necessary to ensure that each ACID transaction is replicated for fault-tolerance. Describing them in detail is out of the scope of this paper, but the replication is performed using a master-slave mechanism over a RoCE (RDMA over Converged Ethernet) network. Unlike traditional cloud storage, we expose a byte-addressable interface that allows libpmem efficient access to NVM while only the transaction records of a sequence of modified objects to be replicated at commit time.

Furthermore, cold data is automatically written to a back-end cloud storage service to reduce the cost of replication – three copies of data in NVM is more expensive than storing data in a replicated cloud storage based on SSDs/HDDs, therefore the replicated NVM tier is used as a cache. This paper focuses on provisioning the NVM required for this cache in the light of imminent NVM diversity.

NVM Diversity Non-volatile memory technologies have matured and are predicted to bring persistence close to the CPU with dramatic improvements in latency [6]. Emerging technologies, such as phase change memory [28, 33], memristors [38], and spin-transfer torque [17, 12, 26] are other promising candidates.

However, some of these technologies have endurance issues and bringing them to market in large volume at low price is still challenging. Prior work on using inexpensive uninterrupted power supply systems to implement battery-backed DRAM (BBRAM) [16] provides non-volatile memory at the speed of DRAM, but has the same capacity and cost concerns as DRAM. However, DRAM has no endurance issues. Table 1 shows the trade-offs between them [39]. Hybrid approaches combine these technologies to obtain "the best of both worlds". However, such an approach is fundamentally different from layering volatile DRAM on top of NVM or layering NVM on top of SSDs as Section 3 will discuss.

3. Hybrid NVM in the Cloud

This paper proposes using two different kinds of byte-addressable persistent memories in combination. One of them, though of smaller capacity, provides higher performance and endurance (fastNVM, e.g. battery-backed DRAM). The other one is based largely on Intel and Micron's 3Xpoint memory but could be any of the alternative memory technologies under development (slowNVM).

This is fundamentally different from tiering volatile DRAM on top of NVM or tiering NVM on top of SSDs. When using volatile DRAM as a cache for NVM, only reads can be accelerated; writes must still reach NVM. When using SSDs as a secondary non-volatile tier, the interface is block oriented and requires a different kind of data management mechanism that may not be relevant for byte-addressable devices. What makes tiering fastNVM on top of slowNVM uniquely challenging is that both are byte-addressable and both are persistent.

More specifically, the actual regions of the memories modified for a given transactional application is fundamentally different when both the tiers are byte-addressable and persistent. This is due to the fact that data can be persistently updated in the fast tier and that the data structures are optimized for byte-addressability unlike when using SSDs. Furthermore, to

Technology	Scaling (not charge based)	Endurance	Latency	Capacity	Cost	Commercially Available	Category
PCM	Yes	$10^6 - 10^8$	100s of ns	<1TB/server	$2-8/GB	Yes	Slow
3D-XPoint	Yes	$10^6 - 10^8$	100s of ns	<1TB/server	$2-8/GB	Yes	Slow
Memristor	Yes	$10^{11} - 10^{12}$	10s of ns	<1TB/server	Potentially lower than NAND	No	Fast
STT-RAM	Yes	$>10^{12}$	10s of ns	<1TB/server	Highest	No	Fast
Battery-backed DRAM	No	10^{16}	10s of ns	<0.5TB/server	$12/GB	Yes	Fast

Table 1: NVM Technologies

the extent of our knowledge, this is the first paper to analyze how durable transactions that leverage byte-addressability execute at an instruction level. The primary difference being that durable transactions need to create durable log entries to ensure atomicity and consistency of the data.

Crucially, the challenges for the cloud provider hosting multiple guests each wanting to tier fastNVM over slowNVM have not been addressed. In this paper, we present a scheduling mechanism that the cloud provider can use to allocate fastNVM and slowNVM resources to customers co-located on shared hardware such that the overall performance increases and the wear reduces on the slowNVM.

Current storage service-level agreements (SLAs) specify minimum bandwidth and/or page-granular IOPS guarantees based on SSDs. However, the benefit of HNVM is to provide a much lower latency service on small granular accesses based on NVMs. As such, we add both average and tail (95% percentile) storage operation access latencies to HNVM's SLA for requests of various sizes. Given a workload profile, our scheduler profiles average and tail latencies for various configurations and suggests the configuration that is optimal from the cost perspective while meeting SLAs.

Finally, we design a workload placement algorithm that avoids SLA violation penalty costs by determining optimal proportions of different NVMs to serve different workloads, mapping them onto servers with fixed resources, and model how sharing of resources across workloads impacts latencies negatively. The placement algorithm extrapolates from workload profiles to simulate how a given combination of workloads would interfere with each other, and uses this information to inform the cloud provider about potential SLA violations.

4. Hybrid NVM Cloud Design

We first describe resources available within a single server, and how VMs are layered on top of servers. We then introduce a cost and resource constraints model to formally specify the workload placement problem. Finally, we present a heuristic that uses workload profiling information to predict sharing behavior of co-located workloads and minimize penalty costs.

4.1. Hybrid NVM in Single Server

As explained in Section 2, NVM is added to compute nodes and a storage cache is implemented on top of it. In our proposal, this NVM is a hybrid between a fastNVM (battery-backed DRAM) and a slowNVM (3D-Xpoint).

Each server has a fixed amount of fastNVM and slowNVM. However, customers receive different number of VMs across various servers with different ratios of slowNVM and fastNVM depending on their agreed-upon SLAs and workloads. For each instance of our accelerated storage service, two additional VMs are instantiated to serve as replicas, which use a fast RDMA network to communicate with the master.

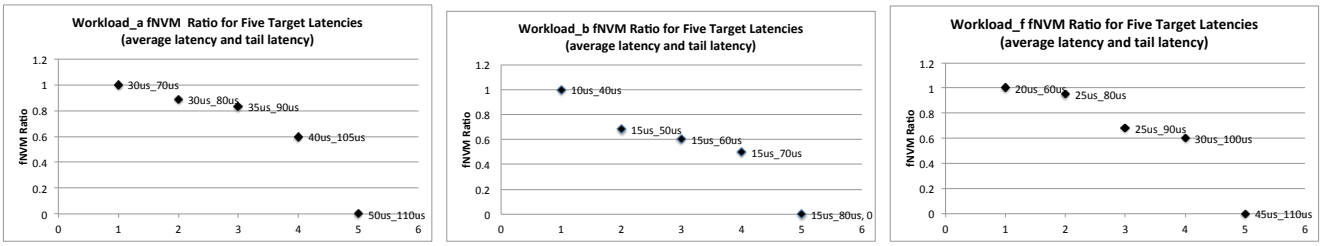
The two NVMs are laid out as a flat memory with system support, hot (most frequently accessed) and highly-written data is stored in the fastNVM to enable faster access and reduce wear of the slowNVM. In today's systems, this support can be implemented by monitoring access and dirty bits in the page table over epochs and moving items appropriately at the beginning of each epoch. Alternatively, we can have a page ranking and migration mechanism in hardware [35, 34].

We have instrumented our compute-side cache running YCSB, a suite of key-value store workloads, and corroborated prior results showing that a small number of pages count for most memory accesses. For example, one YCSB workload of one million operations with *uniformly random* key access and 1KB value size (for a total of over 1GB of data) has 75% of its page accesses in the hottest 100MB of memory (we use this ratio in the results presented in Section 6). If the key accesses follow a more likely *Ziphian* distribution, over 90% of the page accesses hit in the hottest 100MB. This is due to the cache being implemented as a B-tree and values being stored at their leaves. There is still significant data locality as the data structure branches are traversed.

4.2. Optimizing fastNVM and slowNVM Ratio for a Single Workload

Figure 1 shows optimal fastNVM/slowNVM ratios for different target average and tail latencies. The evaluated three workloads (workload_a, workload_b, and workload_f) have totally different fastNVM ratio vs. latencies profile. This implies different applications require different amounts of fastNVM in order to guarantee SLAs.

In order to guarantee storage service-level agreements, a dynamic resource allocation algorithm is developed to adjust the ratio of data in fastNVM and slowNVM during program runtime. In this Algorithm 1, n different configurations are available with increasing ratio of data in fastNVM and slowNVM. The algorithm gradually increases the fastNVM/slowNVM ratio if the any of the achieved latencies (average or tail) is higher than the target one. Otherwise, if the achieved latencies are both some threshold lower than the target latencies, the algorithm decreases the fastNVM/slowNVM ratio. We find the optimal thresholds t_{ave} and t_{tail} that are small enough to avoid resource over-provisioning while still big enough to avoid oscillation.



(a) (b) (c)

Figure 1: Optimal fNVM Ratio for Different Target Latencies (average_tail) a: workload_a; b: workload_b; c: workload_f

Algorithm 1 Find Optimal fastNVM and slowNVM Ratio

Require:

$T_{real\_ave} <= T_{target\_ave}$

$T_{real\_tail} <= T_{target\_tail}$

1: $C_n = initializeConfigurations(n)$ ▷ Initialize n configurations.
2: $C_{current} = C_0$ ▷ Initialize current configuration with least fastNVM.
3: **for** every CONFIG cycles **do**
4: **if** $T_{real\_ave} > T_{target\_ave}$ or $T_{real\_tail} > T_{target\_tail}$ **then**
5: $C_{current} = nextConfig()$ ▷ Find next configuration with higher fastNVM/slowNVM ratio.
6: **else if** $T_{real\_ave} < t_{ave} \times T_{target\_ave}$ and $T_{real\_tail} < t_{tail} \times T_{target\_tail}$ ▷ Latencies are some threshold lower than target latencies. **then**
7: $C_{current} = prevConfig()$ ▷ Find next configuration with lower fastNVM/slowNVM ratio.
8: **end if**
9: **end for**

4.3. Supporting Larger Datasets

For workloads with a dataset larger than a single server's memory can cache, two different mechanisms can be used to provide larger cache space, as shown in Figure 2:

- Data can be sharded into share-nothing partitions across multiple VMs, each with its own separate replica VMs, as described in Section 2. This enables a distribution of data over a large number of machines, with each machine serving a portion of the data. This technique requires multiple VMs but it is highly scalable because these VMs need not interact with each other.
- Remote memory access enables partitioning the data into multiple servers without creating multiple application VM instances. RDMA requests are sent directly to the NIC without involving the kernel and are serviced by the remote NIC without interrupting the CPU. A memory region is registered with the NIC by pinning the pages in physical memory [23].

4.4. NVM-Cache Cost Model

Cloud providers provide worst case guarantees instead of providing exact ones. The reason for this is that applications

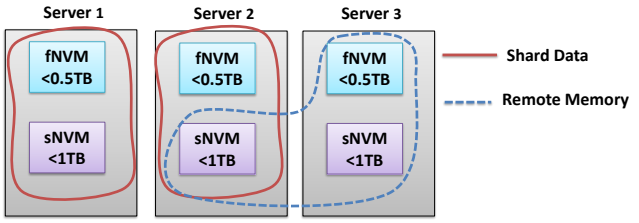


Figure 2: Different Data Placement Mechanisms for Larger Dataset

spanning multiple VMs typically depend on numerous shared software and hardware components. When such guarantees are violated, a provider typically reimburses the customer for the period of time during which the guarantee was violated. We propose a similar approach for the distributed cache based on NVM, but add average and tail latencies (95th percentile) to the SLA.

In this model, a cloud customer and a cloud provider agree on a service level agreement (SLA) that defines the desired performance and the penalty cost for an SLA violation. A cloud customer picks from a set of SLA profiles that a provider is willing to offer. There could be multiple levels of SLAs for key-value stores. The SLA specifies a performance goal in terms of average latency and tail latency at some percentage of requests (e.g., 30μs average and 100μs or less for 95% of requests). In order to enforce the SLA, we also specify a penalty function $p(latency)$. The first part of $p(latency)$ proportionally grows with the difference between the achieved average latency and target average latency. The second part proportionally grows with the total number of requests that exceed the 5% of requests that fall outside the 95% latency. SLA violations may incur revenue loss and should be avoided.

The purpose of the penalty function is to express how much revenue is lost for a cloud provider because of not abiding by the SLA. Average latency and tail latency need to be penalized differently, as average latency affects application overall experience while tail latency affects customers' tolerance to extreme cases. We define Penalty Cost in Equation 1, where T_{real_ave} is the achieved average latency while T_{target_ave} is the target average latency, and N_{real_tail} is the the total number of requests that fall outside the target tail latency (T_{target_tail}) while N_{target_tail} is the target number of request (5% of the

total requests) that fall outside the target tail latency. $p_{ave}(WI)$ (dollars per time unit) and $p_{tail}(WI)$ (dollars per request) are unit penalty costs in terms of workload write density. As write intensive workloads are intrinsically harder to guarantee average latency and tail latency due to network latency and network performance variation, we set unit penalty differently based on workload write intensity.

$$\begin{aligned} PenaltyCost = (T_{real\_ave} - T_{target\_ave}) \times p_{ave}(WI) \\ +(N_{real\_tail} - N_{target\_tail}) \times p_{tail}(WI) \end{aligned} \quad (1)$$

A resource bundle $R_i = [r_{i1}, r_{i2}, r_{i3}, \ldots, r_{in}]$ describes $workload_i$'s (W_i) resource requirements on n types of resources. For HNVM, $R_i = [r_{icpu}, r_{ifnvm}, r_{isnvm}]$ is W_i's resource requirements on three major resources. $R = \begin{bmatrix} R_1 \\ R_2 \\ \ldots \\ R_N \end{bmatrix}$ is a resource matrix for N workloads running across a datacenter.

The resources, however, are constrained by what each server can provide. Each server has only limited hardware resources, such as memory capacity. Thus, each server can support a certain number of workloads. A workload placement vector $p_i = [e_{i1}, e_{i2}, e_{i3}, \ldots, e_{iN}]$ defines if a workload runs on server i. For instance, a value of 1 for e_{i1} means W_1 runs on server i. If there are M servers in the datacenter, a workload placement matrix $P = \begin{bmatrix} p_1 \\ p2 \\ \ldots \\ p_M \end{bmatrix} = [e_1, e_2, \ldots, e_x]$ describes how workloads are placed on x servers. In this matrix, workload placement vector e_i contains only standard basis of $\mathbb{R}^x$ if W_i runs on a single server (e.g., $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ means the workload runs on server 1 only). If W_i requires more resources than a single server can provide, we shard the keys with consistent hashing [25] and place the workload on multiple servers (e.g., $\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$ means the workload runs on server 1 and server 2 with resources evenly distributed).

A resource matrix C describes how much resource is consumed on each server, $C = PR$, where P is a matrix of dimensions MxN where p_{ij} indicates whether a particular workload j out of N has been placed in server i out of M and R is a matrix of dimension NxK where r_{jk} indicates the resource requirement of workload j on resource k.

The problem is to minimize cloud provider's global cost, where cost consists of penalty cost and server operation cost. s.t. $C_{icpu} \leq S(cpu), C_{ifnvm} \leq S(fnvm), C_{isnvm} \leq S(snvm)$, for all server i in M servers, where S is the total resource capacity on a single server. The problem becomes very similar to a conventional bin-packing problem. However, here we minimize cost, instead of number of used bins.

4.5. Workload Placement and Scheduling

The placement algorithm should figure out the best workload placement and scheduling strategy to minimize global cost, as described in section 4.4. While servers have homogeneous hardware configuration with a fixed fastNVM/slowNVM ratio, workloads may use different fastNVM/slowNVM ratios individually. This requires the workload placement algorithm to be aware of server load and how the behaviors of workloads change when resources are shared.

In order to place an incoming workload, the algorithm requires two types of information to guarantee SLA while minimizing cost. First, how does the workload's performance react to the **reserved resources** (e.g. number of cores, memory capacities, etc.)? Reserve resources are such pre-allocated and isolated in current cloud model. Second, how the workload changes its performance profile in face of sharing **best-effort resources**, such as network bandwidth and memory bandwidth contention. Best-effort resources are not provisioned proportional to the payment, thus creating more uncertainty in performance. We propose a HNVM scheduling algorithm to address the above questions.

4.5.1. Profile Workload characteristics can change over time, which can be caused by changes in request rate, request size, request ratio (read/write), etc. In order to address workload characteristic change, we construct a performance profile and define different SLAs for different workload characteristics. The profiler sweeps inputs from all customers with varying request sizes, request rates, and read/write ratios, and creates a single model that will work for any customer **before the entire key-value services starts operating**. Then the profiler recommends possible SLAs between the customer and provider. The cloud provider offers a series of SLAs, each with a particular range of request rates and other workload characteristics, and each for a different price. The customer chooses a SLA based on its workload characteristics. If the client's workload deviates from the contracted SLA, then the cloud provider is not responsible to SLA violations.

The profile maps available best-effort resource (network bandwidth and memory bandwidth) to **achievable operation latency range**. A operation latency range contains an array of achievable average latencies and tail latencies by varying fastNVM and slowNVM ratios. A practical performance range can save five performance points from five different fastNVM and slowNVM ratios. A workload profile can save performance ranges for four network bandwidths(10Gbps, 20Gbps, 30Gbps, and 40Gbps) and four memory bandwidths (8GB/s, 12GB/s, and 24GB/s, 48GB/s). The profile can be less than 5KB in size.

In many cases, only one of the best-effort resources becomes a bottleneck and has a dominating affects on performance. To save profiling time, the profiler can store the sensitivity curve of average latency and tail latency to memory bandwidth, indicating how the workload performance reacts to memory

bandwidth change when network bandwidth is not a bottleneck. Similarly, the profiler constructs a network bandwidth sensitivity curve with full memory bandwidth. With these two curves, the scheduler can predict workload performance when it is co-scheduled with another workload. The available memory bandwidth and network bandwidth on the server indicates which resource is more critical for the new workload. Therefore, the sensitivity curve of the critical resource can by applied to predict the new workload's performance. According to our results, network bandwidth is more critical to write-intensive workloads, and memory bandwidth affects more read-intensive workloads. Performance is very sensitive to the bandwidth when the available bandwidth is less than the required bandwidth of the workload (request rate is high). Both average latency and tail latency increases sharply if bandwidth is further reduced.

4.5.2. Workload Characterization and Scheduling When scheduling a workload, the scheduler first characterizes the incoming workload in order to generate the correct performance profile. In order to characterize a workload, an empty server that runs a local monitor is always reserved. The performance profile is searched with the generated signature of the workload (request rate, request size, and request ratio). The workload profile contains a full mapping of network bandwidths and memory bandwidths to performance ranges. If the workload profile is not cached in the profile, a new workload profile is constructed and inserted.

Figure 3 shows a high-level work flow of the proposed scheduling algorithm. At first, two empty servers characterize two VMs (VM5 and VM6) concurrently. After characterization, the local monitor sends the workload signature (request rate, request size, and request ratio) to the scheduler. If the workload profile is cached in the performance profile, the scheduler determines where to place the workload directly. Otherwise, the scheduler returns a NOT CACHED signal to the local monitor. The local monitor start evaluating performance ranges with different network bandwidths and memory bandwidths and inserts the profile to the global profile.

The scheduler analyzes global resource availability and determines whether to migrate the workload to an non-empty server. The scheduler contains a table of global resource availability (free fastNVM capacity, free slowNVM capacity, network bandwidth, and memory bandwidth) on a per server basis. By mapping any server's available network bandwidth and memory bandwidth to the workload's profile, the scheduler finds the performance range. The optimal fNVM/sNVM ratio is determined by the performance range. Finally, the scheduler migrates VM5 to an occupied server as co-scheduling VM1 and VM5 reduces the global cost. In contrast, VM6 stays on the original server as co-scheduling VM6 with any existing VM increases the global cost. The server where VM6 runs becomes an occupied server.

Different from previous work such as Bubble-up [30] and Bubble-flux [40], HNVM schedules multiple latency-sensitive

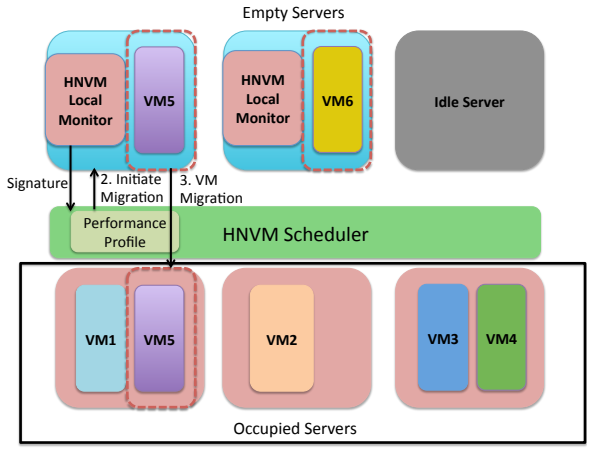


Figure 3: HNVM Scheduler. HNVM local monitor characterizes the incoming workload and sends the signature to the scheduler. The scheduler determines workload performance range by either reading from the performance profile or asking the local monitor to construct a new profile. The scheduler migrates a workload if co-scheduling reduces global cost.

workloads (key-value stores) by proactively predicting and avoiding contention created by best-effort resources. The HNVM Profiler relies on a rate-limiting hardware/software to create a tunable amount of "pressure" on memory system and network stack. Hardware memory traffic shaping [46] can be applied to adjust memory bandwidth on a per-core/per-application basis. Similarly, network interface cards limit network bandwidth for certain channels.

4.5.3. Cost Aware Scheduling We propose a cost-aware scheduler, the HNVM scheduler, that minimizes cloud provider's global cost. Workload consolidation reduces servers in operation by scheduling VMs on a single physical server. However, consolidating workloads creates resource contention, resulting in SLA violations (penalty cost). To address this problem, the HNVM scheduler takes workload performance sensitivity to server load and best-effort resource contention into account while placing workloads.

A conventional bin-packing algorithm does not take into account server penalty cost for SLA violations. A random-fit algorithm randomly picks up a server that provides enough reserved resources (CPUs and memory capacities) that a workload requires. Although efficient, it might not find the optimal server that minimizes cost. A largest-fit algorithm searches for a server with maximal available resources, which results in minimum best-effort resource contention for lower server workloads at the cost of operating more servers. A best-fit algorithm tries to find a server with best resource match with the workload's reserved resource requirements. For example, if the workload requires a resource bundle of x CPU cores, y fast-NVM capacity, and z slow-NVM capacity (x, y, z), a server with best resource match will have least unused resources; $R_{over} = a|X-x| + b|Y-y| + c|Z-z|$; where X, Y, Z are the server's available resources and are greater than x, y,

z. As different resources have different impact on workload performance, different weights are used (a, b, and c) in order to measure the resource match. However, it is not aware of best-effort resource (memory bandwidth and network bandwidth) contention created by co-running VMs. This will potentially cause performance degradation and SLA violations.

The HNVM scheduler uses a cost-aware best-fit scheduling algorithm that minimizes penalty cost caused by SLA violations. The scheduler sorts the candidate servers by their resource availability in decreasing order, favoring servers with high resource utilization at first. Thus the HNVM scheduler packs as many workloads as possible on a single server, but only when the incurred penalty cost is smaller than operation cost of starting a new server. This allows idle servers to be turned down or turned off for power savings. The scheduler predicts the performance impact of the new workload on the existing workloads. It adds the required resources given by HNVM Profiler of the new workload to the consumed resources and maps existing workloads' performance profiles to the new server load. If the scheduler predicts that the introduction of the new workload increases the penalty cost to existing workloads, it starts evaluating the next available server that best matches the resource requirements of the new workload. While searching for the next possible server, the scheduler keeps the smallest penalty cost that has been evaluated so far. If the minimum penalty cost is smaller than the operation cost of starting a new server, the scheduler migrates the workload to the non-empty server that produces the minimum penalty cost. Otherwise, the scheduler leaves the workload on the profiling server.

The scheduler fails on scheduling a new workload if there aren't enough resources available required by the new workload. Admission control does not allow new workloads until there are enough resources or new servers are added to the pool. Figure 4 shows an example of scheduling workloads at different server load levels.

Without losing accuracy, HNVM Profiler saves workload performance profiles at four available network bandwidth levels (10Gbps, 20Gbps, 30Gbps, and 40Gbps) and three memory bandwidth levels (4GB/s, 8GB/s, 12GB/s, and 24GB/s). The HNVM scheduler can map any workload combined with its fastNVM/slowNVM ratio to any server load level and get the predicted performance.

Workload characteristics, however, may change dynamically. Our techniques are relevant even when workloads change dynamically. In the future, we wish to implement an online performance (latency and bandwidth monitor) monitor such that the system can detect if the application SLAs are being violated at runtime. If so, we simply move the workload back into the scheduler after updating the profile of the application with the information collected by the monitor at runtime. We wish to leverage the virtual machine migration mechanisms that cloud systems use today to ease the implementation of such a scheduler that adapts to applicaiton needs dynamically.

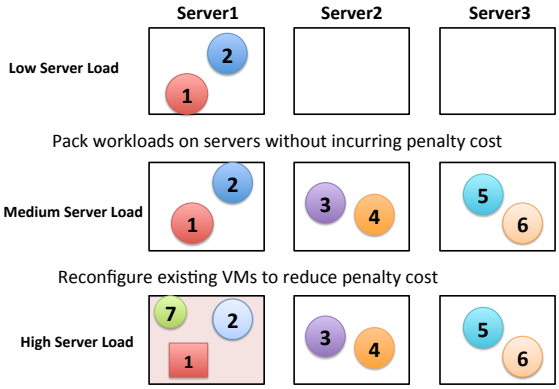


Figure 4: HNVM Scheduling at Different Server Loads. 1. Minimize servers in operation without incurring SLA violations at low to medium server load. 2. Minimize penalty cost by adjusting existing workloads' fast-NVM/slow-NVM ratios at high server load.

Component	Cost ($)	Description
CPU and logic	1000–1500	Dual Socket (12 core each)
Memory	1200–2500	128–256GB
SSD & HDD	700–1200	2–4TB
Network	100–1000	10GigE/RDMA
Other	0–500	power supply, batteries, chassis, etc,.
Total	**3000–6000**	

Table 2: Typical cost of an Open Compute server used by Microsoft and Facebook datacenters.

5. Experimental Setup

We now describe the experimental setup for evaluating the HNVM scheduler and demonstrating how it can reduce the cost for the cloud provider while meeting customer SLAs.

5.1. Hardware Configuration & Cost

Microsoft, Facebook and RackSpace have started using the Open Compute server specifications [8]. The pooled resources of these organizations and an open source model for the specifications enables quick innovation and keeps the costs low for every one because of their combined scale.

We use the Open Compute server specification 2.0 [9] to price our servers based on component costs from NewEgg [7], Amazon, and SuperMicro. Table 2 shows the split. We do note that bulk purchases can reduce the cost by up to 2x as is typical with consumer vs. bulk pricing. The server cost estimate (non-bulk estimate) can be $3–6K depending on the amount of memory and storage. Typically, DRAM's cost is 33–40% of the total cost.

In our system, each server has dual-socket processors of 16 cores. The system has a 40Gbps RDMA network which is used to emulate slower networks. Each server has 144GB of DRAM which is used to emulate both fast-NVM and slow-

NVM. We price the fast-NVM the same as BBRAM (15% higher than DRAM [16]) and slow-NVM the same as 3D-XPoint (10%-20% of DRAM). In our cost model, slow-NVM costs only 10% of fast-NVMM.

5.1.1. NVM Latency Fast-NVM is based on BBRAM that costs 10% higher than DRAM in Azure [16]. We scale up the operation latency distributions measured on a DRAM-based machine to approximate slow-NVM. The AMAT model (Average Memory Access Time) is used to scale up DRAM-based operation latency distributions to distributions with slow-NVM. In order to measure AMAT, we use an Intel Pin Tool to generate memory traces for each key-value store operation. Memory traces are fed into a cache simulator to approximate cache miss rate to calculate AMAT, which is used to scale up the latencies appropriately.

We use two different memory latency profiles for slow-NVM: One with an optimistic read latency of 100ns and write latency of 300ns (similar to average case in NVMDB [39]), and one with more pessimistic read latency of 300ns and write latencies of 1000ns (similar to Mojim [44]). The memory bandwidth for the slow-NVM is 6GB/s (approximately 1/8 of DRAM bandwidth) and for the fast-NVM is 48GB/s (assuming BBRAM).

Overall, the effective latency of hybrid NVM is determined by the page placement policy that the scheduler employs. A page miss rate analyzer runs the memory trace to get page miss rate for a certain sized memory. The analyzer takes a page policy combined with a memory size and outputs the ratio of page hits in the memory. With this analysis, we obtain the ratio of memory accesses in the fast-NVM (e.g.,BBRAM) and the ratio of memory accesses in the slow-NVM (e.g.,PCM) for a given fast-NVM/slow-NVM ratio. However, the latency of remote memory accesses is determined by the network.

5.1.2. Network Latency The default ethernet option in Azure is 10GigE. The RDMA implementation in Azure [18] provides a network that is 8–16X faster. Communication over regular ethernet (10GigE) within compute nodes incurs over 300× lower latency than communication between compute and storage nodes. Note that cloud providers are unlikely to adopt RDMA for reaching storage nodes from compute nodes due to RDMA's high cost (and storage costs need to be low), so the performance gap is expected to increase.

Figure 5 shows how datacenters are organized into separate compute and storage nodes, and latencies that we have **measured** in Azure between these nodes. Our reliable and replicated NVM cache layer at the compute side can significantly reduce the round-trip latency for data that needs to be persisted.

5.2. Simulation and Workload

We developed a trace-driven simulation and analytical tool to evaluate the performance and cost tradeoffs of the proposed system, and investigate the implications on workload placement and scheduling. Five YCSB key-value store workloads are evaluated comprehensively. Figure 6 shows the work flow of our simulation framework. In order to support transactional thread-safe update to the non-volatile memory, we implemented our own key-value store database with the support of the Intel libpmem [14] library. To approximate timing features of slow NVM technologies, such as PCM, we developed a statistic-based analytical flow (Parameter Generation in Figure 6). Finally, a simulation tool Kamino Sim is developed to explore different NVM hierarchies and datacenter design tradeoffs.

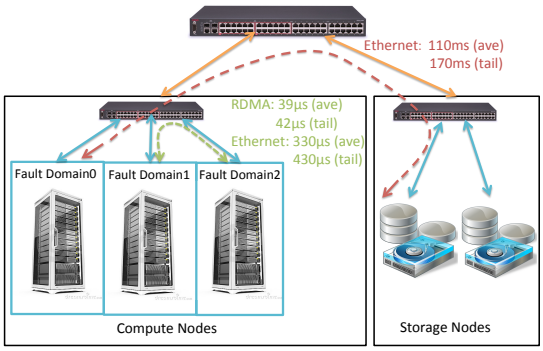


Figure 5: Operation Latencies With and Without NVM Cache at Compute Side

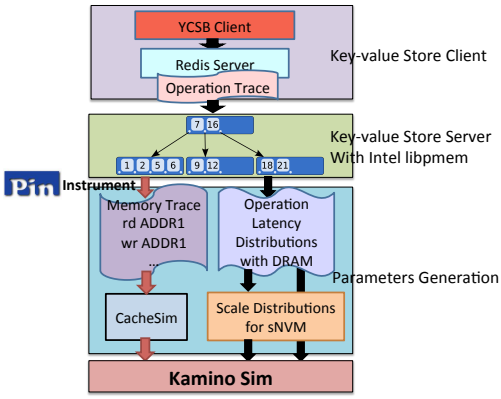


Figure 6: KaminoSim: Simulation Work Flow

5.2.1. Key-value Store Operation Traces Key-value store workload traces are generated by running YCSB [13] on Redis [11]. These five YCSB workloads use a uniform data size of 1KB. We generate traces of YCSB workloads a, b, c, d, and f with one million operations each.

5.2.2. Transactional B-tree Implementation In order to support transactional thread-safe update to the non-volatile memory, we implement a conventional B-tree structure with Intel libpem library. Each transaction is performed in a thread-safe and fail-safe manner via ACID transactions. The B-tree serves as a database server for the key-value store workloads. This B-tree is the key-value store that we instrument using the Intel Pin Tool. We then use the obtained traces to conduct experiments and drive our analysis.

5.2.3. Operation Latency Distribution In order to measure operation latency distributions, we run YCSB workload traces

against the T-tree on a real server machine with 32LB L1, 256KB L2, 10MB L3, and 16GB DRAM. Key-value store operation ("read", "insert", and "update") latency distributions are saved in separate histograms of 10000 bins each. These latency distributions are used for generating latency numbers in the simulator. The network latency and remote update latency are measured based on a real implemention of a non-volatile write-back cache in Microsoft Azure servers.

5.2.4. Monte-Carlo Simulator We developed a Monte-Carlo Simulator to simulate various system aspects, such as cache latencies, and memory and network latencies. The simulator statistically generates operation latencies according to latency distributions described in section 5.2.3. The memory and network bandwidths are limited using a simulated rate limiter.

6. Evaluation

We now evaluate the scheduler using various workloads that are described in the previous section. We first demonstrate how much the lifetime of slow-NVM can be improved because of the scheduler.

6.1. HNVM Lifetime Evaluation

When using a slow-NVM subject to wear issues (e.g., PCM), one of the first design constraints is to provision enough fast-NVM such that the slow-NVM lasts as long as the server, typically 3 years. This section explores worst case wear under various memory performance and network latency characteristics. We assume a slow-NVM endurance (N_{endur}) of 10^6 writes. Note that the write-rate is ultimately determined by the network bandwidth used (ethernet's 10Gbps vs. RDMA's 40Gbps).

We instrument the server responding to the set of YCSB workloads with a Pin [10] Tool including a cache and hybrid NVM model to collect the total number of writes to the slow-NVM. We extract the baseline network latency from a real Azure cloud setup, as shown in Figure 5, and scale it by the appropriate factor. We then feed the data, along with the memory latencies to the Monte-Carlo simulator and derive the total execution time (T_{exec}). We use the analytical model in Equation 2 to determine expected lifetime for a particular memory. If more than one memory type has wear issues, the minimum lifetime should be used, but this is not the case in this study.

$$lifetime = \frac{Size(slow-NVM)}{Size(perWrite) \times \frac{N_s}{T_{exec}}} \times N_{endur} \quad (2)$$

We present the expected memory lifetime for the aggregate of YCSB workloads [13] under different network speeds and fast-NVM/slow-NVM ratios in Figure 7. The numbers are for the faster version of the slow-NVM with 300ns writes. For the same network speed, different fast-NVM/slow-NVM ratios result in different lifetimes. The lifetimes increase until the ratio of fast-NVM to slow-NVM becomes 4:1 because larger fast-NVMs are able to absorb a larger number of hot accesses. However, beyond a certain point, larger fast-NVMs stop helping and start taking away space from slow-NVMs that would have been used for wear leveling. This happens primarily because the number of writes received by the less written pages are surprisingly high enough to wear out the slow-NVM at a faster rate.

For the same fast-NVM/slow-NVM ratio, faster networks require higher ratios of slow-NVM to reach the 3-year server lifetime goal (red line) because increased network performance also increases the number of writes per unit time. For different memory latencies, higher latencies slightly increase memory lifetime. This is because a slower memory means a lower number of writes per time unit. This effect is stronger for configurations with a higher ratio of slow-NVM because more requests are directed toward slow-NVM in such configurations. Overall, to maintain a 3-year server lifetime goal at low network latencies, cloud providers need about 10% fast-NVM when using a slow-NVM with write latency of 300ns, and 4% fast-NVM when using a slow-NVM with write latency of 1000ns.

6.2. Optimal Server Hardware

The server lifetime study above has allowed us to determine minimum amounts of fast-NVM in different scenarios, providing bounds on practical fast-NVM/slow-NVM ratios. Next, we turn our attention to determine the uniform server configuration that optimizes performance and cost, based on the set of workloads to be served.

In Figure 8, performance is measured as the aggregate number of requests per second serviced by a saturated server in that configuration (i.e., adding more threads to the server brings no throughput benefit). Cost is the total cost of that server configuration, factoring in different network interface and NVM prices, as described in Section 5.

Figure 8(a) compares the performance of different workloads under a variety of conditions. Workloads with a higher proportion of writes (a and f) benefit more from improved network performance and improved slow-NVM performance due to faster completion of write request replication. In addition, configurations with proportionally more fast-NVM seem more attractive. By themselves, these results suggest that the server configuration should use the fastest network and only fast-NVM. However, the cost of doing this would be very high.

Figure 8(b) factors cost in, showing performance per cost. This is the real metric a cloud provider would want to optimize when choosing equipment to add to a datacenter. Once again, improved network and memory performance favors workloads with higher proportion of writes (a and f). However, when pushed beyond a certain point, the cost of higher performance parts, both network and memory, becomes too high to justify the added performance, bringing the performance per cost metric down. Moreover, performance per cost is always maximized by hybrid NVM configurations, showing that diversity of NVM parts in datacenters is advantageous.

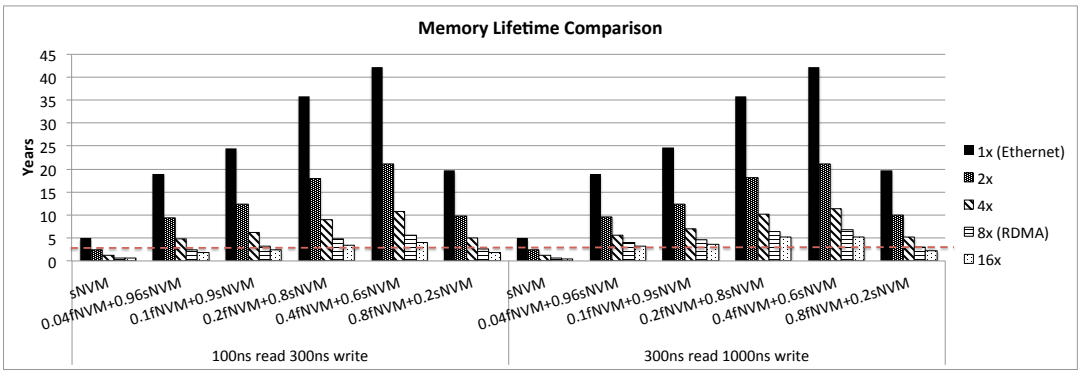
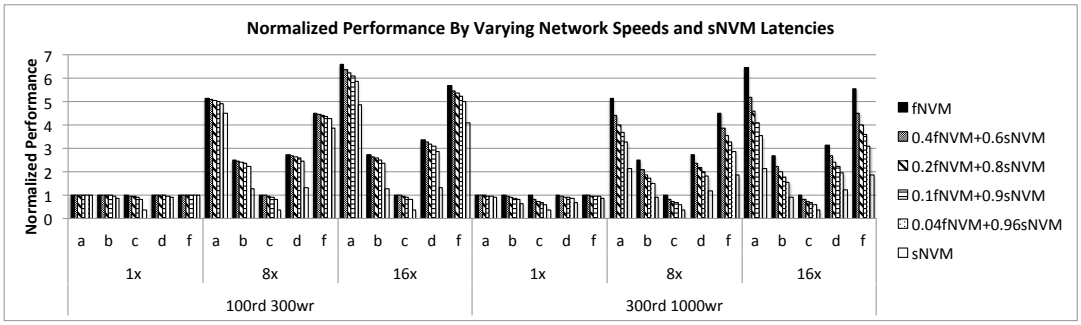
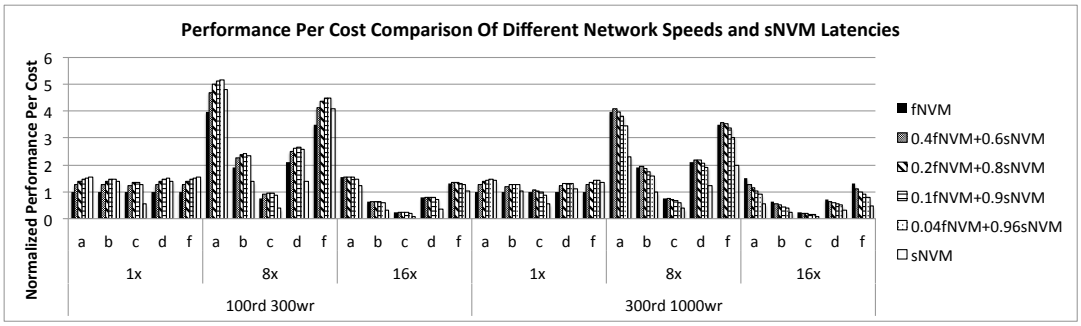


Figure 7: HNVM Lifetime Comparison Under Different Network Speeds and fast-NVM/slow-NVM Ratios. 1x network speed is a typical Ethernet network speed based on the measurement in Microsoft Azure Servers.



(a)



(b)

Figure 8: a. Normalized Performance and b. Normalized Performance Per Cost for Different fast-NVM/slow-NVM Ratios with Three Network Speeds (1x = Ethernet network, 8x = RDMA network, 16x = 100Gbps network).

Overall, the best performance per cost results from the lower latency slow-NVM type, 8× network (RDMA) and 4% fast-NVM. However, based on the analysis in section 6.1, servers will not last the 3-year target with only 4% fast-NVM, thus we use the minimum 10% fast-NVM configuration. Note that servers would also not last the 3-year target with 100% slow-NVM, so this is not a viable single-NVM choice. Compared to the other single-NVM choice, 100% fast-NVM, a hybrid NVM improves performance per cost by 28% for the lower latency slow-NVM. Breaking this down into performance and cost, cost is improved by 33%, while performance is degraded by only 4%, with no violations to the SLA.

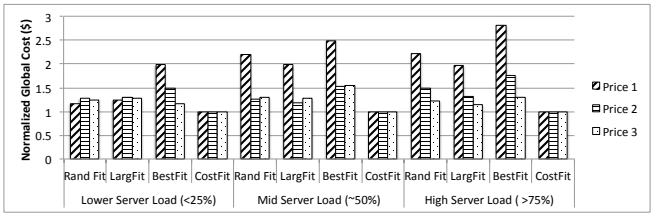


Figure 9: Provider's Global Cost with Four Scheduling Algorithms under Three Price Models (Normalized to CostFit)

6.3. Placement Algorithm Effectiveness

Once servers have been purchased, a provider's goal is to minimize penalty costs with that hardware. A corrolary is also true for first party cloud services run by the cloud provider. Azure

data centers run several first party cloud services including Azure Compute, Azure Storage, Azure DocumentDB, SQL Azure, Azure ML, etc,. It is important for each of these services to reduce their total number of occupied servers while meeting their respective customer demands. Likewise, it is important for the caching service to use as few servers as possible for providing VMs with fast storage as a service to the outside world while using as few Azure resources as possible.

This is accomplished by the proposed cost-aware placement algorithm, CostFit. We compare the global cost of CostFit with three other algorithms: RandFit, which allocates a random server with enough resources required by the workload; LargFit, which allocates the server with maximal available resources, ignoring the workload requirements; and BestFit, which searches for a server with the best resource match to the workload requirements.

Global cost consists of operation cost of all power-up servers and penalty cost for all scheduled VMs. Based on previous datacenter analysis [15], we compute per-server operation cost on a hourly basis. In this experiment, we use three sets of unit penalty costs as described in Function 1.

Figure 9 shows global costs for all algorithms under different server load conditions with the default server configuration as determined in Section 6.2. In Price 1, the provider is penalized more heavily if violating SLAs. In Price 3, powering a server is relatively much more expensive. Under all server conditions, CostFit minimizes the global cost. BestFit always results in worst global cost except in low server utilization with Price 3, where powering up a server is much more expensive. It is because BestFit lacks the awareness of server load vs. workload performance. The diminishing available network bandwidth and memory bandwidth caused by packing more workloads on a single server degrades workload latencies significantly. BestFit cannot predict performance degradation with best-effort resource contentions. Under higher server load, LargFit becomes more effective than RandFit and BestFit. It implies that balancing out workloads across multiple servers is desirable if the scheduler does not understand how resource sharing and contention affects workload performance. Only CostFit takes workload penalty costs under different server loads into account and adjusts VM configurations and migration strategies according to penalty cost.

7. Related Work

Compute-side SSD caching: Recent work has demonstrated the usefulness of compute-side SSD caching [19, 20, 24, 27, 31] in cloud environments to reduce the latency between compute and storage nodes. Our paper extends this work by understanding how hybrid NVMs can be efficiently used to not only further reduce the latency but also to reduce the cost for the provider by providing a resource scheduling mechanism that allocated resources across customers while meeting their performance goals.

Distributed NVM Systems: Distributed in-memory transactional systems, including FaRM [23, 16], Mojim [44], and RamCloud [36] have been proposed to provide data consistency, availability and high performance. While all these designs focus on improving performance and fault tolerance of a distributed system, NVMCloud extends these works to leverage two types of NVMs to provide both high performance and low cost for both cloud providers and customers. It also presents plausible solutions for workload placement and scheduling in a Cloud context.

Utility Based Partitioning and Configuration: Application ulitity based cache partitioning [32, 37, 37] and sub-core configurable processor architecture [45] suggests optimizing hardware partitions and configurations for different workloads sharing the same computation substrate. NVMCloud provides hardware and software mechanisms to configure a combination of BBRAM and PCM, in order to minimize costs while guaranteing SLAs.

Datacenter Workload Scheduling and Optimization: Datacenter workload placement and scheduling [21, 22] that aims to improve server utilization and QoS have been proposed. Heracles [29], Bubble-up, and Bubble-flux [41] manage latency-critical workloads and best-effort workloads by changing hardware and software isolation mechanisms. These scheduling algorithms only co-schedule latency-sensitive workloads with batch workloads. PEGASUS [29] adjusts server power management to meet global service-level latency objectives that improves energy proportionality of WSC systems. Lee's works [43, 42] rethink fairness in computer architecture and propose elastic fairness to find fair allocation of pareto efficiency. In complement to existing techniques, NVMCloud provides SLA guarantees for storage by using a load-aware and cost-aware scheduler.

8. Conclusion

In this work we proposed using multiple NVM technologies to build fast and reliable compute side caches to speed up storage accesses in the cloud. We use a fastNVM in combination with slowNVM to build such a cache. We find that such an architecture can scale arbitrarily while providing up to three orders of magnitude latency reductions over traditional cloud storage. We also find that only a small fraction (4–10%) of NVM is required to be the more expensive fastNVM (battery-backed DRAM)to ensure that the less durable slowNVM (PCM) lasts as long as the server does for typical cloud storage workloads. We propose a new resource allocation and scheduling algorithm to use different proportions of NVM while meeting latency and capacity requirements for co-located applications. Our proposed design saves providers' cost by 33%. The proposed scheduling algorithm reduces providers' total cost by more than 20%, 50%, and 50% at three server loads.

References

[1] Amazon elastic block store. https://aws.amazon.com/ebs/.

[2] Amazon elastic compute cluster. https://aws.amazon.com/ec2/.

[3] Amazon simple storage service. https://aws.amazon.com/s3/.

[4] Azure compute. https://azure.microsoft.com/en-us/services/virtual-machines/.

[5] Azure storage. https://azure.microsoft.com/en-us/services/storage/.

[6] Intel non-volatile memory. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[7] Newegg. newegg.com.

[8] Open compute project. http://www.opencompute.org/.

[9] Open compute server spec. 2.0. http://www.opencompute.org/projects/.

[10] Pin: A dynamic binary instrumentation tool. http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[11] Redis. http://redis.io.

[12] Spin memory shows its might. http://spectrum.ieee.org/semiconductors/memory/spin-memory-shows-its-might.

[13] Yahoo cloud serving benchmark. https://labs.yahoo.com/news/yahoo-cloud-serving-benchmark.

[14] pmem.io: Persistent memory programming, 2015. http://pmem.io/nvml/libpmem/.

[15] Dave Maltz Albert Greenberg. What goes into a data center - sigmetrics 2009 tutorial, June 2009.

[16] Dragojevic Aleksandar, Narayanan Dushyanth, Nightingale Edmund, Renzelmann Matthew, Shamis Alex, Badam Anirudh, and Castro Miguel. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70, 2015.

[17] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *J. Emerg. Technol. Comput. Syst.*, 9(2):13:1–13:35, May 2013.

[18] Azure RDMA. http://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available/.

[19] Deepavali Bhagwat, Mahesh Patil, Michal Ostrowski, Murali Vilayannur, Woon Jung, and Chetan Kumar. A Practical Implementation of Clustered Fault Tolerant Write Acceleration in a Virtualized Environment. In *Proc. 13th USENIX FAST*, Santa Clara, CA, February 2015.

[20] Steve Byan, James Lentini, Anshul Madan, Luis Pabon, Micheal Condict, Jeff Kimmel, Steve Kleiman, Christopher Small, and Mark Storer. Mercury: Host-side Flash Caching for the Data Center. In *Proc. 28th IEEE MSST*, Pacific Grone, CA, April 2012.

[21] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[22] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[24] David A. Holland, Elaine Angelino, Geideon Wald, and Margo L. Seltzer. Flash Caching on the Storage Client. In *Proc. USENIX ATC*, San Jose, CA, June 2013.

[25] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[26] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno. 2mb spin-transfer torque ram (spram) with bit-by-bit bidirectional current write and parallelizing-direction current read. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 480–617, Feb 2007.

[27] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write-Policies for Host-side Flash Caches. In *Proc. 11th USENIX FAST*, San Jose, CA, February 2013.

[28] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, January 2010.

[29] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.

[30] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM.

[31] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable Writeback for Client-side Flash Caches. In *Proc. USENIX ATC*, Philadelphia, PA, June 2014.

[32] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, 2006.

[33] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[34] L. Ramos and R. Bianchini. Exploiting phase-change memory in cooperative caches. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 227–234, Oct 2012.

[35] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[36] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, 2014. USENIX.

[37] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 57–68, New York, NY, USA, 2011. ACM.

[38] Dmitr Strukov, Gregory Snider, Duncan Stewart, and Stantley Williams. The missing memristor found. *Nature*, 453:80–83, May 2008.

[39] Kosuke Suzuki and Steven Swanson. The non-volatile memory technology database (nvmdb). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015. http://nvmdb.ucsd.edu.

[40] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM.

[41] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM.

[42] Seyed Majid Zahedi and Benjamin C. Lee. Ref: Resource elasticity fairness with sharing incentives for multiprocessors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 145–160, New York, NY, USA, 2014. ACM.

[43] S.M. Zahedi and B.C. Lee. Sharing incentives and fair division for multiprocessors. *Micro, IEEE*, 35(3):92–100, May 2015.

[44] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

[45] Yanqi Zhou and David Wentzlaff. The sharing architecture: Sub-core configurability for iaas clouds. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 559–574, New York, NY, USA, 2014. ACM.

[46] Yanqi Zhou and David Wentzlaff. Mitts: Memory inter-arrival time traffic shaping. In *Proceedings of the 43th Annual International Symposium on Computer Architecture*, ISCA, 2016.