# Memtrade: Marketplace for Disaggregated Memory Clouds

HASAN AL MARUF, University of Michigan, USA
YUHONG ZHONG, Columbia University, USA
HONGYI WANG, Columbia University, USA
MOSHARAF CHOWDHURY, University of Michigan, USA
ASAF CIDON, Columbia University, USA
CARL WALDSPURGER, Carl Waldspurger Consulting, USA

We present *Memtrade*, the first practical marketplace for disaggregated memory clouds. Clouds introduce a set of unique challenges for resource disaggregation across different tenants, including resource harvesting, isolation, and matching. Memtrade allows producer virtual machines (VMs) to lease both their unallocated memory and allocated-but-idle application memory to remote consumer VMs for a limited period of time. Memtrade does not require any modifications to host-level system software or support from the cloud provider. It harvests producer memory using an application-aware control loop to form a distributed transient remote memory pool with minimal performance impact; it employs a broker to match producers with consumers while satisfying performance constraints; and it exposes the matched memory to consumers through different abstractions. As a proof of concept, we propose two such memory access interfaces for Memtrade consumers – a transient KV cache for specified applications and a swap interface that is application-transparent. Our evaluation using real-world cluster traces shows that Memtrade provides significant performance benefit for consumers (improving average read latency up to 2.8×) while preserving confidentiality and integrity, with little impact on producer applications (degrading performance by less than 2.1%).

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Virtual machines**; **Virtual memory**; **Distributed memory**; **Allocation / deallocation strategies**; • **General and reference** → **Performance**; **Evaluation**;

Additional Key Words and Phrases: Memory Disaggregation, Memory Harvesting, Remote Memory Marketplace

**ACM Reference Format:**
Hasan Al Maruf, Yuhong Zhong, Hongyi Wang, Mosharaf Chowdhury, Asaf Cidon, and Carl Waldspurger. 2023. Memtrade: Marketplace for Disaggregated Memory Clouds. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 2, Article 41 (June 2023), 27 pages. https://doi.org/10.1145/3589985

## 1 INTRODUCTION

Cloud resources are increasingly being offered in an elastic and disaggregated manner. Examples include serverless computing [8, 13] and disaggregated storage [5, 72, 81, 82, 114] that scale rapidly and adapt to highly dynamic workloads [2, 83, 84, 86, 97]. Memory, however, is still largely provisioned statically, especially in public cloud environments. In public clouds, a user launching a new VM typically selects from a set of static, pre-configured instance types, each with a fixed number of cores and a fixed amount of DRAM [4, 29, 40]. Although some platforms allow users to customize the amount of virtual CPU and DRAM [19], the amount remains static throughout the lifetime of

2476-1249/2023/6-ART41 $15.00
https://doi.org/10.1145/3589985

the instance. Even in serverless frameworks, which offer elasticity and auto-scaling, a function has a static limit on its allocation of CPU and memory [9].

At the same time, long-running applications deployed on both public and private clouds are commonly highly over-provisioned relative to their typical memory usage. For example, cluster-wide memory utilization in Google, Alibaba, and Meta datacenters hovers around 40%–60% [74, 102, 104, 108]. Large-scale analytics service providers that run on public clouds, such as Snowflake, fare even worse – on average 70%–80% of their memory remains unutilized [114]. Moreover, in many real-world deployments, workloads rarely use all of their allocated memory all of the time. Often, an application allocates a large amount of memory but accesses it infrequently (§2.2). For example, in Google's datacenters, up to 61% of allocated memory remains idle [85]. In Meta's private datacenters, within a 10-minutes window, applications use only 30–60% of the allocated memory [96]. Since DRAM is a significant driver of infrastructure cost and power consumption [21, 67, 68, 96], excessive underutilization leads to high capital and operating expenditures, as well as wasted energy (and carbon emissions). Although recent remote memory systems address this by satisfying an application's excess memory demand from an underutilized server [47, 51, 56, 74, 85, 95, 103], existing frameworks are designed for private datacenters.

Furthermore, with the emergence of coherent interfaces like Compute Express Link (CXL) [18], next-generation datacenter designs are moving towards tiered-memory subsystems [89, 96]. Servers within a rack can be connected through CXL switches and access each other memory. In such a system, CPUs of one server will have access to heterogeneous memory types with varied latency, bandwidth, and performance characteristics (Figure 1). While running multiple applications in such a rack-scale system, system-wide application-level performance will highly depend on an application's share to different memory tiers. Efficiently rightsizing different memory tiers, moving cold pages from faster to slower memory tiers, and matching harvested memories on different tiers to appropriate applications with a view to ensuring performance is challenging in such a disaggregated system.

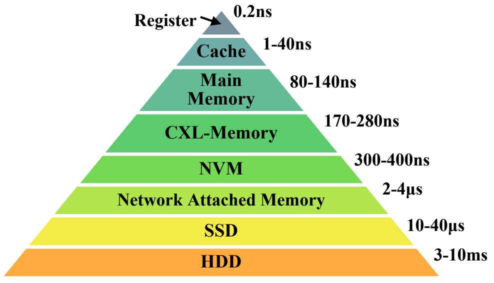


Fig. 1. Latency characteristics of different memory technologies in a disaggregated system.

In this paper, we harvest both unallocated and allocated-but-idle application memory to enable remote memory in public clouds. We propose a new memory consumption model that allows over-provisioned and/or idle applications (*producers*) to offer excess idle memory to memory-intensive applications (*consumers*) that are willing to pay for additional memory for a limited period of time at an attractive price, via a trusted third-party (*broker*). Participation is voluntary, and either party can leave at any time. Practical realization of this vision must address following challenges:

(1) *Immediately Deployable.* Our goal is that Memtrade is immediately deployable on existing public or private clouds. Prior frameworks depend on host kernel or hypervisor modifications [47, 51, 74, 85, 95, 108]. In many cases (e.g., public cloud setting, server configuration restrictions), a tenant cannot modify host-level system software which would require the operator to manage the remote memory service. In addition, prior work assumes the latest networking hardware and protocols (e.g., RDMA) [47, 56, 74, 85, 95, 103, 108]; availability of these features in public clouds is limited, restricting adoption. Memtrade being an ubiquitous solution, allows users hop

on any existing machine and readily deploy it without disrupting the running application(s) and modifying the underline kernel.

(2) *Efficient Harvesting.* Memory harvesting needs to be lightweight, transparent and easily deployable without impacting performance. Most prior work includes only a VM's *unallocated* memory in the remote memory pool. Leveraging idle application-level memory – allocated to an application but later unused or accessed infrequently – significantly enhances remote memory capacity. This is especially challenging in public clouds, where a third-party provider has limited visibility of tenant workloads, and workloads may shift at any time. Existing *cold page detection*-based [44, 85] proactive page reclamation techniques need significant CPU and memory resources, along with host kernel or hypervisor modifications [96].

(3) *Performant Consumption.* To ensure producer-side performance, Memtrade must return a producer's harvested memory seamlessly when needed. Memory offered to consumers may also disappear due to a sudden burst in the producer's own memory demand, or if a producer leaves unexpectedly. Memtrade needs to manage this unavailability to provide a high-performance memory interface. Besides, modern datacenters are going to observe an architectural paradigm shift towards memory pooling and disaggregation where efficiently migrating hot/cold memories to appropriate memory tiers will be a major concern [89, 96]. Memtrade harvester needs to be aware of the impact of different memory tiers. Effectively harvesting free spaces on the faster memory tiers while moving colder pages to the slower tiers can help tiered-memory subsystems.

(4) *Incentivization and Resource Matching.* Unlike prior work, which assumes cooperative applications, in a public cloud setting, we need to create a *market* where producers and consumers have monetary incentives to participate. Producers must be compensated for leasing memory, and the price must be attractive to consumers compared to alternatives (e.g., existing in-memory caching services or spot instances). In addition, producers have varied availability and bursty workload demands, while consumers may have their own preferences regarding remote memory availability, fragmentation, network overhead, and application-level performance, all which must be considered when matching producers to consumers.

We design and develop *Memtrade*, an immediately-deployable realization of remote memory on public clouds that addresses these challenges without any host kernel or hypervisor modifications. Memtrade employs a *harvester* in each producer VM to monitor its application-level performance and adaptively control resource consumption.The harvester uses an adaptive control loop that decides when to harvest from and when to return memory to the producer.

To prevent performance degradation in memory-sensitive applications, we design a novel in-memory swap space, *Silo*, which serves as a temporary victim cache for harvested pages. In the case of a sudden loss of performance, the harvester proactively prefetches previously-harvested pages back into memory. The combination of these mechanisms allows Memtrade to harvest idle pages with low impact to producer workload performance and offer them to consumers. Consumers of Memtrade can access the harvested memory through a key-value (KV) cache or a swap interface. For public cloud adaptation or secure memory consumption, Memtrade consumer interfaces provide optional plugable cryptographic protections for the confidentiality and integrity of data stored in the untrusted producer VM.

To maximize cluster-wide utilization, Memtrade employs a *broker* – a central coordinator that manages the remote-memory market and matches producers and consumers based on their supply and demand, and helps facilitate their direct communication. The broker sets the price per unit of remote memory and is incentivized by receiving a cut of monetary transactions. The broker does not require any support from the cloud provider, and it does not need to economically benefit the cloud providers to be viable. Of course, cloud providers may choose to support it, but it's optional.

One can implement their own pricing strategies based on different objectives; Memtrade broker service is orthogonal towards the pricing and matching mechanisms.

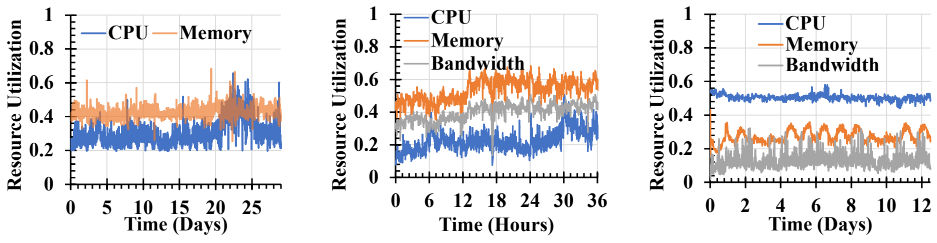
Although we focus primarily on virtualized public clouds, Memtrade can be deployed in other settings, such as private datacenters and containerized clouds. We plan to open-source Memtrade. Overall, we make the following contributions:

- Memtrade is the first end-to-end system that enables a marketplace for disaggregated memory on clouds (§3). Memtrade is easily-deployable without any support from the cloud provider.
- We design a system to identify and harvest idle memory with minimal overhead and negligible performance impact (§4), which uses Silo – a novel in-memory victim cache for swapped-out pages to reduce performance loss during harvesting an application's idle memory.
- We design a broker that arbitrates between consumers and producers, enables their direct communication and implements a placement policy based on consumer preferences, fragmentation, and producer availability (§5).
- Memtrade improves consumer average latency by up to 2.8×, while impacting producer latency by less than 2.1% (§7), and improves memory utilization (up to 97.9%).

## 2 BACKGROUND AND MOTIVATION

### 2.1 Remote Memory

Remote memory exposes capacity available in remote hosts as a pool of memory shared among many machines. It is often implemented logically by leveraging unallocated memory in remote machines via well-known abstractions, such as files [47], remote memory paging [51, 56, 71, 74, 90], distributed OS virtual memory management [108] and the C++ Standard Library data structures [103]. Existing frameworks require specialized kernels, hypervisors or hardware that might not be available in public clouds. Prior works focus on private-cloud use cases [47, 51, 56, 74, 85, 103] and do not consider the transient nature of public-cloud remote memory, nor the isolation and security challenges when consumers and producers belong to different organizations.

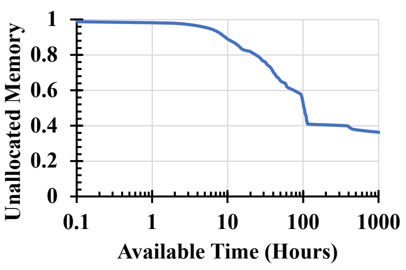


(a) **Google Cluster** (b) **Alibaba Cluster** (c) **Snowflake Cloud**

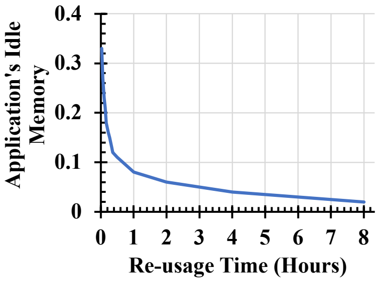
Fig. 2. Cluster resources remain significantly unallocated in (a) Google, (b) Alibaba, and (c) Snowflake.

### 2.2 Resource Underutilization in Cloud Computing

*Underutilized Resources.* Due to over-provisioning, a significant portion of resources remains idle in private and public clouds that run a diverse mix of workloads. To demonstrate this, we analyze production traces of Google [25], Alibaba [1], and Snowflake [114] clusters for periods of 29 days, 36 hours, and 14 days, respectively (Figure 2). In Google's cluster, averaging over one-hour windows, memory usage never exceeds 60% of cluster capacity. In Alibaba's cluster, at least 30% of the total memory capacity always remains unused. Even worse, in Snowflake's cluster, which runs on public clouds, 80% of memory is unutilized on average.

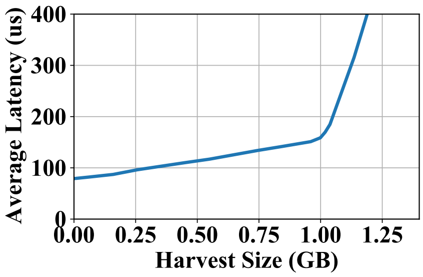


(a) **Google Cluster Memory**

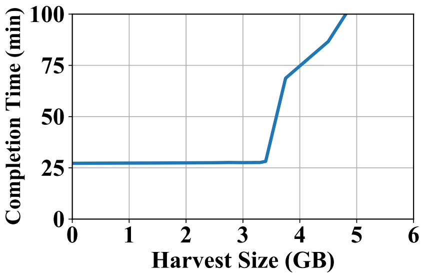


(b) **Idle Application Memory**

Fig. 3. (a) Unallocated memory remains available for long periods, but (b) idle memory are reused quickly.



(a) **Zipfian on Redis**



(b) **XGBoost**

Fig. 4. Performance drop while harvesting memory for (a) Zipfian trace on Redis, and (b) XGBoost training.

However, idle memory alone is not sufficient for providing remote memory access; in the absence of dedicated hardware such as RDMA, it also requires additional CPU and network-bandwidth both at the consumer and the producer. Fortunately, the production traces show that a significant portion of these resources are underutilized. Approximately, 50–85% of cluster CPU capacity remains idle in all of these traces; Alibaba and Snowflake traces, which include bandwidth usage, show that 50–75% of network capacity remains idle.

*Availability of Unallocated and Idle Memory.* Another important consideration is whether unutilized memory remains available for sufficiently long periods of time to enable other applications to access it productively. Figure 3a shows that 99% of the unallocated memory in the Google cluster remains available for at least an hour. Beyond unallocated memory, which constitutes 40% of the memory in the Google traces, a significant pool of memory is allocated to applications but remains idle [51]. Figure 3b shows that an additional 8% of total memory is application memory that is not touched for an hour or more. In public clouds, where many tenants are over-provisioned, the proportion of application idle memory may be much higher [114].

*Uses for Transient Remote Memory.* Transient remote memory seems attractive for numerous uses in many environments. KV caches are widely used in cloud applications [42, 53, 55, 62, 98, 122], and many service providers offer popular in-memory cache-as-a-service systems [6, 12, 30, 34]. Similarly, transient memory can be used for filesystem-as-a-service [126] in serverless computing. Application developers routinely deploy remote caches in front of persistent storage systems, and in-memory caches are a key driver of memory consumption in clouds [53, 98].

## 2.3 Disaggregation Challenges in Public Clouds

*Harvesting Application Memory.* Beyond unallocated memory, a large amount of potentially-idle memory is allocated to user VMs. In many cases, harvesting such idle memory has minimal performance impacts. However, harvesting too aggressively can result in severe performance degradation, or even crash applications. Figure 4 shows the performance degradation while harvesting memory from two applications. We can harvest a substantial amount of memory from each without much performance loss. However, performance can quickly fall off a cliff, and dynamic application load changes necessitate adaptive harvesting decisions in real-time.

To reclaim an application's idle memory, existing solutions use kernel modifications [44, 85, 96] to determine the age of pages mapped by the application. A well-explored technique is to periodically scan the *accessed bit* present in page table entries to infer if a physical page is accessed in a given time period. If a page's age goes beyond a threshold, then it is marked as a cold page and, therefore, reclaimed. Such an approach is difficult to deploy in public clouds, where each user controls its own kernel distribution. Moreover, continuous page tracking can consume significant CPU and memory and require elevated permissions from the host kernel [27].

*Transience of Remote Memory.* Producers and consumers have their own supply and demand characteristics. At the same time, producers may disappear at any time, and the amount of unallocated and idle memory that can be harvested safely from an application varies over time. Given the remote I/O cost, too much churn in memory availability may deteriorate a consumer's performance.

*Security.* VMs that do not belong to the same organization in the public cloud are completely untrusted. Since consumer data residing in remote memory may be read or corrupted due to accidents or malicious behavior, its confidentiality and integrity must be protected. Producer applications must also be protected from malicious remote memory operations, and the impact of overly-aggressive or malicious consumers on producers must be limited.

## 3 MEMTRADE: OVERVIEW

*Memtrade* is a system that realizes remote memory on existing clouds. It consists of three core components (Figure 5): **(i) producers**, which expose their harvested idle memory to the remote-memory market (§4); **(ii) the broker**, which pairs producers with consumers while optimizing cluster-wide objectives, such as maximizing resource utilization (§5); and **(iii) consumers**, which request remote-memory allocations based on their demand and desired performance characteristics (§6). This section provides an overview of these components and their interactions; more details appear in subsequent sections.

*Producers.* A producer employs a collection of processes to harvest idle memory within a VM, making it available to the remote-memory market. A producer voluntarily participates in the market by first registering with the broker. The producer then monitors its resource usage and application-level performance metrics, periodically notifying the broker about its resource availability. The producer harvests memory slowly until it detects a possible performance degradation, causing it to back off and enter recovery mode. During recovery, memory is returned to the producer application proactively until its original performance is restored. When it is safe to resume harvesting, the producer transitions back to harvesting mode.

When the broker matches a consumer's remote memory request to the producer, it is notified with the consumer's connection credentials and the amount of requested memory. The producer then exposes harvested memory through fixed-sized slabs dedicated to that consumer. A producer may stop participating at any time by deregistering with the broker.

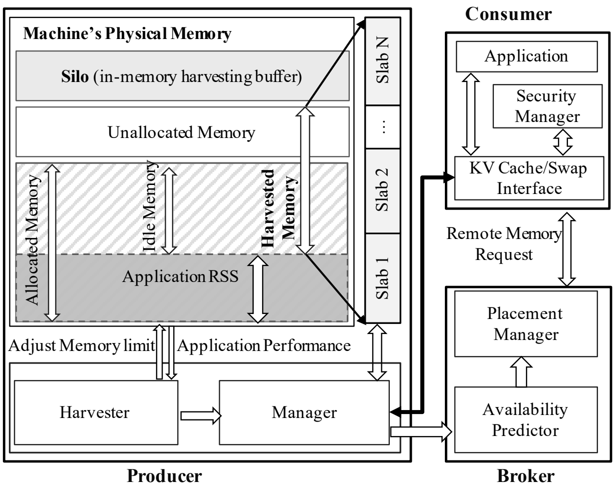


Fig. 5. Memtrade architecture overview.

*Broker.* The broker arbitrates between producers and consumers, matching supply and demand for harvested remote memory while considering consumer preferences and constraints. While Memtrade supports untrusted producers and consumers from diverse tenants, its logically-centralized broker component should be run by a trusted third party – such as a caching-as-a-service provider [30, 34] or the public cloud operator. The broker facilitates the direct connection between the consumer and producer using existing virtual private cloud interconnection tools (e.g.AWS Transit Gateway), which allow the consumer and producer to communicate even if they are on separate virtual private networks [11, 24]. Note that cloud providers already support this capability. The broker decides on the per-unit remote memory price for a given lease time, based on monitoring the current price of spot instances offered in the same public cloud. Appropriate pricing provides incentives for both producers and consumers to participate in the market; the broker receives a cut of the monetary transactions it brokers as commission.

To improve the availability of transient remote memory, the broker relies on historical resource usage information for producers to predict their future availability. It additionally considers producer reputations, based on the frequency of any prior broken leases, in order to reduce occurrences of unexpected remote memory revocations. Finally, it assigns producers to consumers in a manner that maximizes the overall cluster-wide resource utilization.

*Consumers.* A consumer voluntarily participates in the remote-memory market by registering its connection credentials with the broker. Once approved by the broker, the consumer can submit a remote memory request by specifying its required remote memory, lease time, and preferences. After matching the request with one or more producers, the broker sends a message to the consumer with connection credentials for the assigned producer(s).

The consumer then communicates directly with assigned producers through a simple KV cache `GET` / `PUT` / `DELETE` interface to access remote memory. We also implement a transparent remote-paging interface for the consumer. When memory will be occasionally evicted by producers, and a the data needs to be stored persistently, a swap interface may face performance issues. Conveniently, applications using caches assume that data is not persistent, and may be evicted asynchronously. To ensure confidentiality and integrity of consumer data stored in producer memory, consumer interfaces offer an optional cryptographically access to remote memory in a transparent manner (§6.1).

**Algorithm 1** Harvester Pseudocode

```
1: procedure DoHarvest
2:   Decrease cgroup memory limit by ChunkSize
3:   sleep(CoolingPeriod)                          ▷ wait for performance impact
4: procedure DoRecovery
5:   while RecoveryPeriod not elapsed do
6:     Disable cgroup memory limit
7: procedure RunHarvester
8:   for each performance monitor epoch do
9:     if no page-in then
10:      Add performance data point to baseline estimator
11:      Generate baseline performance distribution
12:    Generate recent performance distribution
13:    if performance drop detected then
14:      DoRecovery()
15:    else
16:      DoHarvest()
17:    if severe performance drop detected then
18:      Prefetch from disk
```

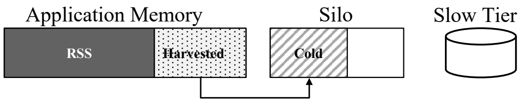
## 4 PRODUCER

The producer consists of two key components: the *harvester*, which employs a control loop to harvest application memory, and the *manager*, which exposes harvested memory to consumers as remote memory. The producer does not require modifying host-level software, facilitating deployment in existing public clouds. Our current producer implementation only supports Linux VMs. The harvester coordinates with a loadable kernel module within the VM to make harvesting decisions, without recompiling the guest kernel.

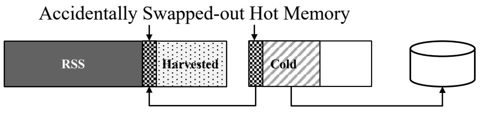
The harvester runs producer applications within a Linux control group (cgroup) [15] to monitor and limit the VM's consumption of resources, including DRAM and CPU; network bandwidth is managed by a custom traffic shaper (§4.2). Based on application performance, the harvester decides whether to harvest more memory or release already-harvested memory. Besides unallocated memory which is immediately available for consumers, the harvester can increase the free memory within the VM by reducing the resident set size (RSS) of the application. In *harvesting mode*, the cgroup limit is decreased incrementally to reclaim memory in relatively small chunks; the default `ChunkSize` is 64 MB. If a performance drop is detected, the harvester stops harvesting and enters *recovery mode*, disabling the cgroup memory limit and allowing the application to fully recover.

Because directly reclaiming memory from an application address space can result in performance cliffs if hot pages are swapped to disk, we introduce *Silo*, a novel in-memory region that serves as a temporary buffer, or victim cache, holding harvested pages before they are made available as remote memory. Silo allows the harvester to return recently-reclaimed pages to applications efficiently. In addition, when it detects a significant performance drop due to unexpected load, Silo proactively prefetches swapped-out pages from disk, which helps mitigate performance cliffs. Algorithm 1 presents a high-level sketch of the harvester's behavior.

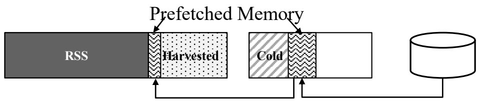
The manager exposes the harvested memory via a key-value cache `GET` / `PUT` / `DELETE` interface, by simply running a Redis server for each consumer. A key challenge for the manager is handling the scenario where the harvester needs to evict memory. We leverage the existing Redis LRU cache-eviction policy, which helps reduce the impact on consumers when the producer suddenly needs more memory.



(a) **Silo first swaps to faster memory tier.**



(b) **Keeps hot pages in faster memory, evicts cold pages to slower memory tier.**



(c) **Proactive prefetching during bursts.**

Fig. 6. Memory harvesting and recovery using Silo.

## 4.1 Adaptive Harvesting of Remote Memory

*Monitoring Application Performance.* The harvester provides an interface for applications to periodically report their performance, with a metric such as latency or throughput. Without loss of generality, our description uses a performance metric where higher values are better. Many applications already expose performance metrics. For example, applications often periodically write their throughput, latency, network bandwidth usage, etc. metrics to a log structure. From this information, we can directly understand the performance variations overtime. For applications with standard interface for monitoring performance metrics, the harvester can simply leverage it. Otherwise, the harvester uses the swapped-in page count (promotion rate) as a proxy for performance [85]. Besides, other Linux features (e.g., Pressure-Stall Information [37]) also provide insights on an application's resource usage. One can leverage those information and correlate them to the application's performance metric. All of these approaches are transparent to the applications – Memtrade doesn't need to modify the application to monitor its performance.

*Estimating the Baseline.* To determine whether the memory limit should be decreased or increased, the harvester compares the current application performance metric to baseline values observed *without memory harvesting*. Of course, measuring performance without memory harvesting is difficult while the producer is actively reclaiming memory. To estimate the baseline performance without harvesting, we use statistics for swap-in events. When there are no swap-in events, the application has enough memory to run its workload. Therefore, the harvester includes the performance metric collected during these times as a baseline. An efficient AVL-tree data structure is used to track these points, which are discarded after an expiration time. Our current implementation adds a new data point every second, which expires after a 6-hour `WindowSize`. We found this yielded good performance estimates; shorter data-collection intervals or longer expiration times could further improve estimates, at the cost of higher resource consumption (§7.1).

*Detecting Performance Drops.* To decide if it can safely reduce the cgroup memory limit, the harvester checks whether performance has degraded more than expected from its estimated baseline performance. Similar to baseline estimation, the harvester maintains another AVL tree to track application performance values over the same period.

After each performance-monitoring epoch, it calculates the 99th percentile (p99) of the recent performance distribution. The harvester assumes performance has dropped if the recent p99 is worse than baseline p99 by `P99Threshold` (by default, 1%), and it stops reducing the cgroup size, entering a recovery state. It then releases harvested memory adaptively to minimize the performance drop. Different percentiles or additional criteria can be used to detect performance drops.

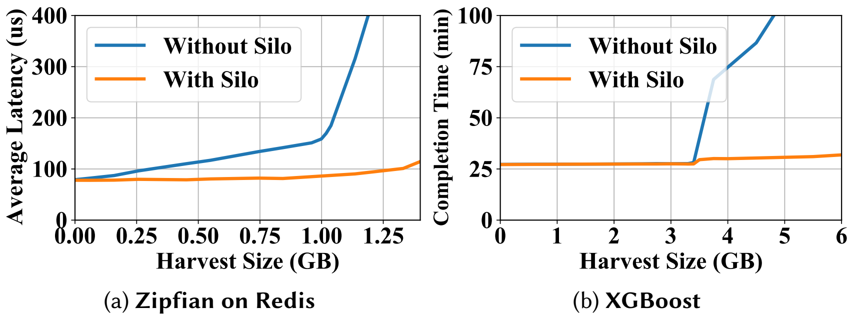


Fig. 7. Performance degradation caused by harvesting different amounts of memory with and without Silo for (a) Zipfian trace running on Redis, and (b) XGBoost training.

*Effective Harvesting with Silo.* The harvester reclaims memory until a performance drop is detected. However, some workloads are extremely sensitive, and losing even a small amount of hot memory can result in severe performance degradation. Also, because the harvester adjusts the application memory size via a cgroup, it relies on the Linux kernel's Page Frame Reclamation Algorithm (PFRA) to make decisions. Unfortunately, PFRA is not perfect and sometimes reclaims hot pages, even with an appropriate memory limit.

To address these problems, we design *Silo*, a novel in-memory area for temporarily storing swapped-out pages. We implement Silo as a loadable kernel module that is a backend for the Linux frontswap interface [22]. The guest kernel swaps pages to Silo instead of disk, thus reducing the cost of swapping (Figure 6a). If a page in Silo is not accessed for a configurable `CoolingPeriod` (by default, 5 minutes), it is evicted to a slower memory tier (which is disk in a traditional server setup). Otherwise, an access causes it to be efficiently mapped back into the application's address space (Figure 6b). In effect, Silo is an in-memory victim cache, preventing hot pages from being swapped to slow memory tier. Figure 7 shows that Silo can prevent performance cliffs, allowing the harvester to avoid significant performance degradation.

When the harvester initially causes some pages to be swapped out to Silo, it cannot determine the performance impact until the `CoolingPeriod` ends. As a result, it harvests cautiously, monitoring the application's RSS, which is available in the memory stats for its cgroup. If it enters harvesting mode, triggering the PFRA to move some pages to Silo, then the harvester refrains from further decreasing the cgroup limit for at least the `CoolingPeriod`. Afterwards, if performance is still stable, the harvester may resume decreasing the limit. This ensures that the harvester will not explore the harvesting size too aggressively without considering the performance impact caused by any disk I/O.

*Handling Workload Bursts.* Simply disabling the cgroup memory limit may not prevent performance drops in the face of sudden bursts. Memtrade addresses this issue by prefetching previously-reclaimed pages, proactively swapping them in from slow memory tiers. If the current performance is worse than all the recorded baseline data points for consecutive epochs, the harvester instructs Silo to prefetch `ChunkSize` of the most recently swapped-out pages (Figure 6c). Producers with a low tolerance for performance degradation and compressible data could alternatively use a compressed RAM disk [43] instead of a disk-based swap device. This would provide more rapid recovery, trading off total harvestable memory.

### 4.2 Exposing Remote Memory to Consumers

The manager communicates with the broker to report resource availability, and it exposes a KV or swap interface to consumers. The harvested memory space is logically partitioned into fixed-size slabs; a slab (by default, 64 MB) is the granularity at which memory is leased to consumers. Different slabs from the same producer can be mapped to multiple consumers for performance and load balancing. Upon receiving a message from the broker, the manager create a lightweight producer store in the producer VM, dedicated to serving remote memory for that consumer.

In Memtrade, the producer store is implemented by running a Redis [33] server within a cgroup in the producer VM, providing a familiar KV cache interface to consumers. Since an empty Redis server consumes only 3 MB of memory and negligible CPU, for simplicity, the manager runs a separate producer store for each consumer. The producer can limit the maximum CPU used by producer stores via cgroup controls. However, producer-side CPU consumption is typically modest; YCSB on Redis uses 3.1% of a core on average.

When the lease period expires, before terminating the Redis server, the manager checks with the broker to determine if the consumer wants to extend its lease (at the current market price). Otherwise, the producer store is terminated and slabs are returned to the remote memory pool.

*Network Rate Limiter.* The manager limits the amount of network bandwidth used by each consumer. We implemented a standard token-bucket algorithm [80] to limit consumer bandwidth. The manager periodically adds tokens to each consumer bucket, in proportion to its allotted bandwidth specified in the consumer request for remote memory. Before serving a request, the producer store checks the consumer's available token count; if the I/O size exceeds the number of available tokens, it refuses to execute the request and notifies the consumer.

*Eviction.* The size of a producer store is determined by the amount of memory leased by each consumer. Once a producer store is full, it uses the default Redis eviction policy, a probabilistic LRU approximation [101]. In case of sudden memory bursts, the manager must release memory back to the producer rapidly. In this scenario, the harvester asks the manager to reclaim an aggregate amount of remote memory allocated to consumers. The manager then generates per-consumer eviction requests proportional to their corresponding producer store sizes and employs the approximate-LRU-based eviction for each producer store.

*Defragmentation.* The size of KV pairs may be smaller than the OS page size [57, 61, 98], which means that an application-level eviction will not necessarily free up the underlying OS page if other data in the same page has not been evicted. Fortunately, Redis supports memory defragmentation, which the producer store uses to compact memory.

## 5 BROKER

The Memtrade broker is a trusted third-party that facilitates transactions between producers and consumers. It can be operated by the cloud provider, or by another company that runs the market as a service, similar to existing caching-as-a-service providers [30, 34]. Producers and consumers participate in the remote-memory market voluntarily, by registering their respective credentials with the broker. Each producer periodically sends its resource utilization metrics to the broker, which uses the resulting historical time series to predict future remote-memory availability over requested lease periods. Consumers request remote memory by sending allocation requests to the broker with the desired number of slabs and lease time, along with other preferences such as acceptable latency and bandwidth. The broker connects producers and consumers that may reside in separate virtual private clouds (VPCs) via existing high-bandwidth peering services [24, 41]. The broker maps consumer requests to producer slabs using an assignment algorithm that satisfies consumer preferences, while minimizing producer overhead and ensuring system-wide wellness objectives (e.g., load balancing and utilization).

In our current implementation, the broker runs on a single node and can handle a market with thousands of participating VMs (§7.2). Since consumers communicate directly with assigned producers until their leases expire, even if the broker is temporarily unavailable, the system can still continue to operate normally, except for the allocation of new remote memory. For higher availability, the broker state could be replicated using distributed consensus, e.g., leveraging Raft [20, 100] or ZooKeeper [78]. The Memtrade operator may also run several broker instances, each serving a disjoint set of consumers and producers (e.g., one broker per region or datacenter).

## 5.1 Availability Predictor

Remote memory is transient by nature and can be evicted at any time to protect the performance of producer applications. Hence, allocating remote memory without considering its availability may result in frequent evictions that degrade consumer performance. Fortunately, application memory usage often follows a predictable long-term pattern, such as exhibiting diurnal fluctuations [65]. The broker capitalizes on historical time series data for producer memory consumption, predicting the availability of offered remote memory using an Auto Regressive Integrated Moving Average (ARIMA) model [76]. Producers with completely unpredictable usage patterns are not suitable for Memtrade. ARIMA model parameters are tuned daily via a grid search over a hyperparameter space to minimize the mean squared error of the prediction.

## 5.2 Remote Memory Allocation

*Constraints and Assumptions.* While matching a consumer's remote memory request, the broker tries to achieve the aforementioned goals under the following assumptions:

1. *Online requests:* Consumers submit remote memory requests in an online manner. During a placement decision, new or pending requests may be queued.
2. *Uncertain availability:* It is unknown exactly how long producer remote memory slabs will remain available.
3. *Partial allocation:* The broker may allocate fewer slabs than requested, as long as it satisfies the minimum amount specified by the consumer.

*Placement Algorithm.* When the broker receives an allocation request from a consumer, it checks whether at least one producer is expected to have at least one slab available for the entire lease duration (§5.1), at a price that would not exceed the consumer budget (§5.3). The broker calculates the placement cost of the requested slabs based on the current state of all potential producers with availability, as a weighted sum of the following metrics: number of slabs available at a given producer, predicted availability (based on ARIMA modeling), available bandwidth and CPU, network latency between the consumer and producer, and producer reputation (fraction of remote memory not prematurely evicted during past lease periods – premature eviction of consumer data hurts a producer's reputation). A consumer may optionally specify weights for each of these placement desirability metrics with its request.

The broker selects the producer with the lowest placement cost, greedily assigning the desired number of slabs. If the producer cannot allocate the entire request, the broker selects the producer with the next-lowest cost and continues iteratively until there are no slabs left to allocate or no available producers. When fewer than the requested number are allocated, a request for the remaining slabs is appended to a queue. Pending requests are serviced in FIFO order until they are satisfied, or they are discarded after a specified timeout.

### 5.3 Remote Memory Pricing

Remote memory must be offered at a price that is attractive to both producers and consumers, providing incentives to participate in the remote memory market. Considering the transient nature of harvested memory, any monetary incentive for leasing otherwise-wasted resources is beneficial to a producer, provided its own application-level performance is not impacted. This incentive can help the producer defray the expense of running its VM. A consumer must weigh the monetary cost of leasing remote memory against the cost of running a static or spot instance with larger memory capacity. In case if a rack-scale tiered memory system within any private cloud, the goal for the broker can be better resource utilization while maintaining the service-level agreement (SLA) for each applications.

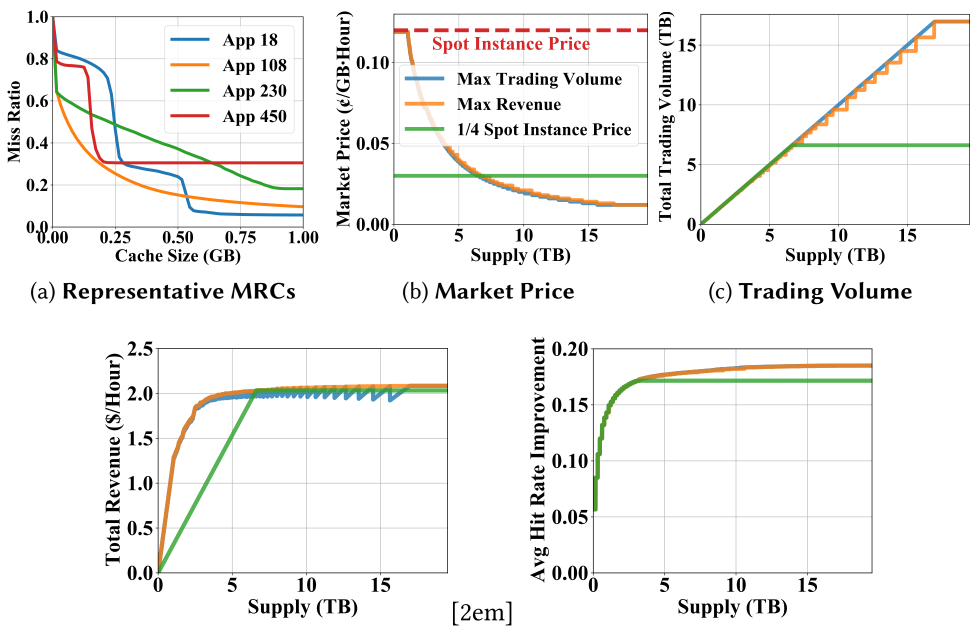
In a public cloud setup, the broker sets a price for leasing a unit of remote memory (GB/hour) and makes it visible to all consumers. Various economic objectives could be optimized (e.g., total trading volume, total revenue of producers, etc.) If the primary goal is improving cluster-wide utilization, the broker needs to focus on maximizing the total trading volume that can be achieved by setting a market-clearing price for remote memory. In this regards, the broker can continuously monitor the price of different instance types in the same public cloud. Specifically, for a given lease, the broker can set the per-unit remote memory price to a value below that associated with the lowest available instance price at that time. On the other hand, if the broker wants to maximize its cut of the revenue, it will try to optimize the total revenue of producers. By default, we assume, the broker wants to maximize its profit. Although, one can come up with any complex market dynamics and plug in their customized policies to achieve the broker's goal or market economics as a whole. Memtrade provides the interface to operate on any pricing model to address different economic objectives. It's up to its users to choose the appropriate policy.

From a consumer's perspective, an alternative to Memtrade is running a separate spot instance and consuming its memory remotely [118]. Thus, to be economically viable to consumers, the price of remote memory in Memtrade should never exceed the corresponding spot instance price. For simplicity, the broker initially sets the price for each unit of remote memory to one quarter of the current market price for a spot instance, normalized by its size. This initial lower price makes remote memory attractive to consumers. Afterwards, the price is adjusted to approximate the maximal total producer revenue by searching for a better price locally. In each iteration, the broker considers the current market price $p$, $p + \Delta p$, and $p - \Delta p$ as the candidates for the price in the next iteration, where $\Delta p$ is the step size (by default, 0.002 cent/GB·hour). Then the broker chooses the one that generates the maximal total revenue for producers. Our pricing model yields good performance in real-world traces (§5.4). Of course, alternative price-adjustment mechanisms can be designed to achieve different economic objectives. One can simply add on their own pricing model and economic objectives to the broker and enjoy the benefit of Memtrade.

### 5.4 Pricing Strategy

To study the impact of pricing strategy on the market, we simulate several pricing strategies with different objectives, such as maximizing the total trading volume, and maximizing the total revenue of producers. Our baseline sets the remote memory price to one quarter of the current spot-instance price (Figure 8b). We simulate 10,000 consumers that use Memtrade as an inexpensive cache. To estimate the expected performance benefit for consumers, we select 36 applications from the MemCachier trace, generate their MRCs (Figure 17), and randomly assign one to each consumer. For each consumer, we ensure that local memory serves at least 80% of its optimal hit ratio.

The strategies that maximize total trading volume (Figure 8c) and total producer revenue (Figure 8d) both adjust the market price dynamically to optimize their objectives. Significantly, when



(a) **Representative MRCs** (b) **Market Price** (c) **Trading Volume**

[2em]

(d) **Producer Revenue** (e) **Consumer Performance**

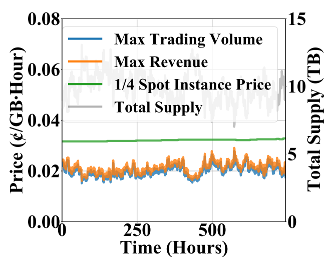
Fig. 8. Effects of different pricing strategies. Revenue and trading-volume maximization both perform similarly.

the remote-memory supply is sufficient, all three pricing strategies can improve the relative hit ratio for consumers by more than 16% on average (Figure 8e).

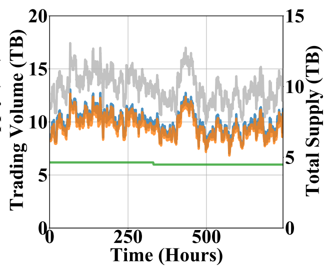
We also examine temporal market dynamics using a real-world trace to simulate a total supply of remote memory which varies over time. We use the idle memory statistics from the Google Cluster Trace 2019 – Cell C [25] to generate the total supply for each time slot, and assume one Google unit represents 5 GB of memory. For the spot instance price, we use the AWS historical price series of the spot instance type r3.large in the region us-east-2b [10]. Figure 9 plots the results. Consistent with the earlier pricing-strategy results in Figure 8, the max-trading-volume and max-revenue strategies effectively adjust the market price based on both supply and demand (Figure 9a). The behavior of all the three pricing strategies with different economic objectives show similar levels of consistency (Figure 9b–9d).

We also consider a more realistic scenario where consumers consider the probability of being evicted when using MRCs to calculate their demand. If the eviction probability is 10%, the total revenue will decrease by 7.6% and 7.1% with the max-trading-volume strategy and the max-revenue strategy, respectively. Also, the cluster-wide utilization reductions corresponding to the max-trading-volume strategy and the max-revenue strategy are 0.0% and 5.8% relatively.

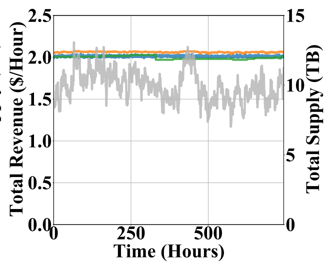
In practice, the broker may have no prior knowledge regarding the impact of market price on consumer demand. In this case, we adjust the market price by searching for a better price locally with a predefined step size (0.002 cent/GB·hour). Figure 9e demonstrates the effectiveness of this approach using a simulation with the Google trace; the market price deviates from the optimal one by only 3.5% on average. Cluster-wide utilization increases from 56.8% to 97.9%, consumer hit ratios improve by a relative 18.2%, and the consumer's cost of renting extra memory reduces by an average of 82.1% compared to using spot instances.



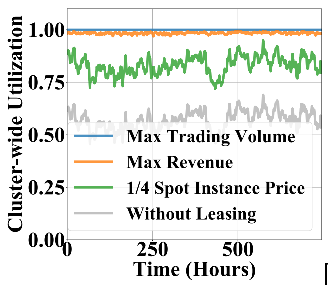
(a) **Market Price**



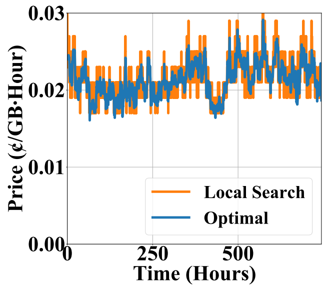
(b) **Trading Volume**



(c) **Producer Revenue**



[2em]



(d) **Cluster Utilization** (e) **Price Adjustment**

Fig. 9. Temporal market dynamics with simulated supply time-series from Google Cluster Trace 2019.

## 6 CONSUMER

A consumer uses remote memory. It first sends its remote memory demand to the broker, based on the current market price given by the broker and its expected performance benefit. After receiving an assignment from the broker, the consumer communicates with producers directly during the lease period. To ensure the confidentiality and integrity of its remote data, the consumer employs standard cryptographic methods during this communication. Rate-limiting techniques are used to protect producers from misbehaving or malicious consumers.

The consumer can use either a KV cache or a swap interface, which we built on top of Redis [33] and Infiniswap [74] clients, respectively. By default, Memtrade uses the key-value interface, because in contrast to swapping to disk, applications using a KV cache naturally assume cached data can disappear. We have also found the KV interface performs better than the swap interface (§7.3), due to the added overhead of going through the block layer when swapping. For the sake of brevity, we focus our description on the KV interface.

### 6.1 Confidentiality and Integrity

This section explains how the consumer ensures data confidentiality and integrity during its KV operations. The subscripts $C$ and $P$ are used to denote consumer-visible and producer-visible data, respectively.

*PUT Operations.* To perform a PUT, the consumer prepares a KV pair $(K_C, V_C)$ to be stored at a remote producer. First, the value $V_C$ is encrypted using the consumer's secret key and a fresh, randomly-generated initialization vector ($IV$). The $IV$ is prepended to the resulting ciphertext, yielding the value $V_P$ to be stored at the producer. Next, a secure hash $H$ is generated for $V_P$, to verify its integrity and defend against accidental or malicious corruption by the producer.

To avoid exposing the lookup key $K_C$, the consumer substitutes a different key $K_P$. Since $K_P$ need only be unique, it can be generated efficiently by simply incrementing a counter for each new key stored at a producer. The producer store storing the KV pair can be identified using an index $P_i$ into a small table containing producer information.

The consumer stores the metadata tuple $M_C = (K_P, H, P_i)$ locally, associating it with $K_C$. While many implementations are possible, this can be accomplished conveniently by adding $(K_C, M_C)$ to a local KV store, where an entry serves as a proxy for obtaining the corresponding original value. This approach also enables range queries, as all original keys are local.

*GET Operations.* To perform a `GET`, the consumer first performs a local lookup using $K_C$ to retrieve its associated metadata $M_C$, and sends a request to the producer using substitute key $K_P$. The consumer verifies that the value $V_P$ returned by the producer has the correct hash $H$; if verification fails, the corrupted value is discarded. The value $V_P$ is then decrypted using $IV$ with the consumer's encryption key, yielding $V_C$.

*DELETE Operations.* To perform a consumer-side eviction, the consumer first removes the metadata tuple $M_C$ from its local store. It then sends an explicit `DELETE` request to the respective producer store so that the consumer and producer store contents remain synchronized.

*Metadata Overhead.* In our current prototype, each consumer uses a single secret key to encrypt all values. Encryption uses AES-128 in CBC mode, and hashing uses SHA-256, both standard constructions. By default, the integrity hash is truncated to 128 bits to save space. A 64-bit counter is employed to generate compact producer lookup keys. The resulting space overhead for the metadata $M_C$ corresponding to a single KV pair $(K_C, V_C)$ is 24 bytes; the $IV$ consumes an additional 16 bytes at the producer.

For applications where consumer data is not sensitive, value encryption and key substitution are unnecessary. Such an integrity-only mode requires only the integrity hash, reducing the metadata overhead to 16 bytes.

### 6.2 Purchasing Strategy

A consumer must determine a cost-effective amount of memory to lease to meet its application-level performance goals. In general, it may be difficult to estimate the monetary value of additional memory. However, when its application is a cache, lightweight sampling-based techniques [77, 115, 117] can estimate miss ratio curves (MRCs) accurately, yielding the expected performance benefit from a larger cache size.

The consumer estimates the value of additional cache space using the current *price-per-hit* from the known cost of running its VM, and its observed hit rate. The expected increase in hits is computed from its MRC, and valued based on the per-hit price. When remote memory is more valuable to the consumer than its cost at the current market price, it should be leased, yielding an economic consumer surplus.

## 7 EVALUATION

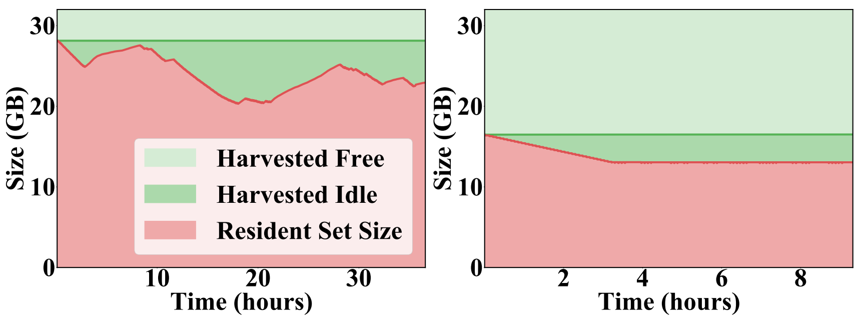
We evaluate Memtrade on a CloudLab [17] cluster using both synthetic and real-world cluster traces.[1] Our evaluation addresses the following questions:

- How effectively can memory be harvested? (§7.1)
- How well does the broker assign remote memory? (§7.2)
- What are Memtrade's end-to-end benefits? (§7.3)

[1]Memtrade can be readily deployed on any major cloud provider. We run our evaluation in CloudLab since it is free.

| | Total Harvested | Idle Harvested | Workload Harvested | Perf Loss |
|---|---|---|---|---|
| Redis | 3.8 GB | 23.7% | 17.4% | 0.0% |
| memcached | 8.0 GB | 51.4% | 14.6% | 1.1% |
| MySQL | 4.2 GB | 21.7% | 7.0% | 1.6% |
| XGBoost | 18.3 GB | 15.4% | 17.8% | 0.3% |
| Storm | 3.8 GB | 1.1% | 1.4% | 0.0% |
| CloudSuite | 3.6 GB | 2.5% | 15.3% | 0.0% |

Table 1. Total memory harvested (idle and unallocated), the percentage of idle memory, the percentage of application-allocated memory, and the performance loss of different workloads.



(a) **MemCachier on memcached** (b) **Training classifier on XGBoost**

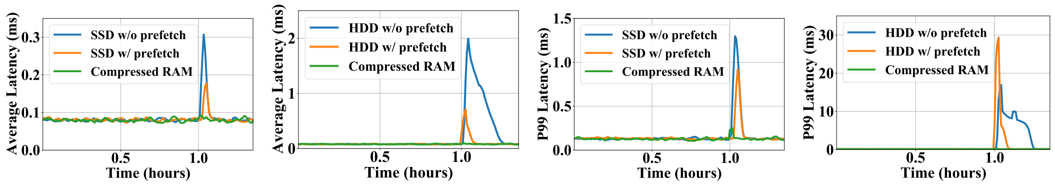
Fig. 10. VM memory composition over time. Additional results are provided in the appendix as Figure 16 for the interested.

*Experimental Setup.* Unless otherwise specified, we configure Memtrade as follows. The producer averages application-level latency over each second as its performance metric. We generate both the baseline performance distribution and the recent performance distribution from data points over the previous 6 hours (`WindowSize`). If the recent p99 drops below the baseline p99 by more than 1% (`P99Threshold`), it is considered a performance drop. Harvesting uses a 64 MB `ChunkSize` and a 5-minute `CoolingPeriod`. If a severe performance drop occurs for 3 consecutive epochs, Silo prefetches `ChunkSize` from disk.

Each physical server is configured with 192 GB DRAM, two Intel Xeon Silver 4114 processors with 20 cores (40 hyperthreads), and a 10Gb NIC. We use Intel DC S3520 SSDs and 7200 RPM SAS HDDs. We run the Xen hypervisor (v4.9.2) with Ubuntu 18.04 (kernel v4.15) as the guest OS.

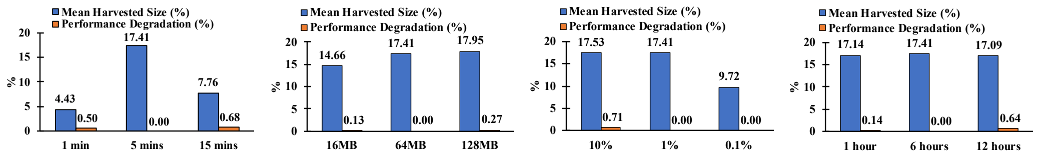
*Workloads.* Consumers run YCSB [64] on Redis [33]. Producers run the following applications and workloads:

- **Redis** running a Zipfian workload using a Zipfian constant of 0.7 with 95% reads and 5% updates.
- **memcached** and **MySQL** running MemCachier [30, 60] for 36 hours and 40 hours, respectively. We use 70 million `SET` operations to populate the memcached server, followed by 677 million queries for memcached and 135 million queries for MySQL.
- **XGBoost** [58] training an image classification model on images of cats and dogs [16] using CPU, with 500 steps.
- **Storm** [36] running the Yahoo streaming workload [59] for 1.5 hours.
- **CloudSuite** [26] executing a web-serving benchmark with memcached as the cache and MySQL as the database, with 1000 users and 200 threads.



(a) **SSD Swap Device** (b) **HDD Swap Device** (c) **SSD Swap Device** (d) **HDD Swap Device**

Fig. 11. Prefetching enables faster recovery and better latency during workload bursts. Memory compression enables even faster recovery, trading off the total amount of harvestable memory.



(a) **Silo `CoolingPeriod`** (b) **Harvesting `ChunkSize`** (c) **`P99Threshold`** (d) **Monitor `WindowSize`**

Fig. 12. **Sensitivity analysis for the harvester.** Single-machine experiments using Redis with YCSB Zipfian constant 0.7.

*VM Rightsizing.* To detemine the VM size for each workload, we find the AWS instance type [3] with the minimal number of cores and memory that can fit the workload without affecting its baseline performance. We use configurations of M5n.Large (2 vCPU, 8 GB RAM) for Redis, M5n.2xLarge (8 vCPU, 32 GB RAM) for memcached and XGBoost, C6g.2xLarge (8 vCPU, 16 GB RAM) for MySQL, C6g.xLarge (4 vCPU, 8 GB RAM) for Storm, C6g.Large (2 vCPU, 4 GB RAM) for CloudSuite, and T2.xLarge (4 vCPU, 16 GB RAM) for consumer YCSB.

## 7.1 Harvester

*Effectiveness.* To observe the effectiveness of the harvester, we run the workloads with their respective producer configurations. For Redis, memcached, and MySQL we use average latency to measure performance. Since XGBoost, Storm, and Cloudsuite do not provide any real-time performance metric, we use the promotion rate (number of swapped-in pages) as a proxy for performance.

We find that Memtrade can harvest significant amounts of memory, even from right-sized VMs (Table 1). Here, a notable portion of the total harvested memory is extracted from the application's idle memory (on average, 1.1–51.4% across the entire workload) at a lower performance degradation cost of 0–1.6%. Also, a whole-machine, all-core analysis shows that the producer-side CPU and memory overheads due to the harvester were always less than 1%.

Figure 10 plots memory allocation over time for two representative workloads, and shows that for workloads such as MemCachier with varying access patterns (Figure 10a), the harvester dynamically adjusts the amount of harvested memory. For most workloads, the percentage of idle harvested memory is higher at the end of the run. Therefore, we expect that if we ran our workloads longer, the average percentage of idle harvested memory would only increase.

*Impact of Burst-Mitigation Techniques.* To observe the effectiveness of the harvester during workload bursts, we run YCSB on Redis using a Zipfian distribution (with constant 0.7). To create a workload burst, we abruptly shift it to a uniform distribution after one hour of the run. Figure 11 shows the average latency and p99 latency using different burst-mitigation approaches. When enabled, Silo prefetches harvested pages back into memory, which helps the application reduce its recovery time by 28.6% and 65.5% over SSD and HDD, respectively. Although prefetching causes a higher peak p99 latency on HDD due to contending with swapping I/O, this high tail latency is transient and the application can recover much faster. A compressed RAM disk exhibits minimal performance drop period during the workload burst (68.7% and 10.9% less recovery time over prefetching from SSD and HDD, respectively), at the cost of less harvested memory.

*Sensitivity Analysis.* We run YCSB over Redis with a Zipfian constant of 0.7 to understand the effect of each parameter on harvesting and producer performance. Each parameter is evaluated in an isolated experiment. Figure 12 reports the results, using average performance to quantify degradation.

The Silo `CoolingPeriod` controls the aggressiveness of harvesting (Figure 12a). Setting it too high leads to less harvesting, while setting it too low causes performance drops that eventually also leads to less harvested memory.

Both the harvesting `ChunkSize` (Figure 12b) and the `P99Threshold` (Figure 12c) affect harvesting aggressiveness in a less pronounced way. The amount of harvested memory increases with more aggressive parameters, while the performance impact always remains below 1%. The performance-monitoring `WindowSize` does not significantly change either the harvested memory or performance (Figure 12d).

## 7.2 Broker Effectiveness

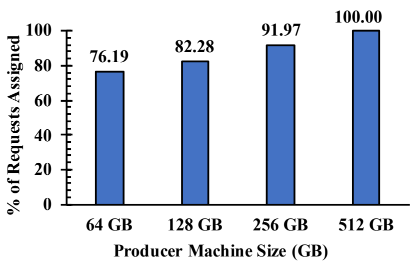
To evaluate the effectiveness of the broker, we simulate a disaggregated memory consumption scenario by replaying two days worth of traces from Google's production cluster [25]. Machines with high memory demand – often exceeding the machine's capacity – are treated as consumers. Machines with medium-level memory pressure (at least 40% memory consumption throughout the trace period) are marked as producers. When a consumer's demand exceeds its memory capacity, we generate a remote memory request to the broker. We set the consumer memory capacity to 512 GB, the minimum remote memory slab size to 1 GB, and a minimum lease time of 10 minutes. In our simulation, 1400 consumers generate a total of 10.7 TB of remote memory requests within 48 hours. On the producer side, we simulate 100 machines.

On average, the broker needs to assign 18 GB of remote memory to the producers per minute. Our greedy placement algorithm can effectively place most requests. Even for a simulation where producers have only 64 GB DRAM, it can satisfy 76% of the requests (Figure 13a). With larger producers, the total number of allocations also increases. As expected, Memtrade increases cluster-wide utilization, by 38% (Figure 13b).

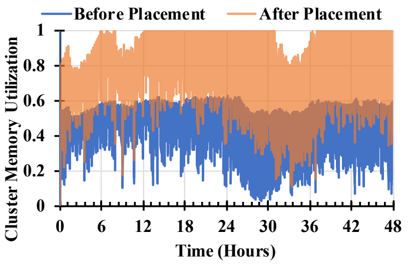
*Availability Predictor.* To estimate the availability of producer memory, we consider its average memory usage over the past five minutes to predict the next five minutes. ARIMA predictions are accurate when a producer has steady usage or follows some pattern; only 9% of the predicted usage exceeds the actual usage by 4%. On average, 4.59% of the allocated producer slabs get revoked before their lease expires.

## 7.3 Memtrade's Overall Impact

*Encryption and Integrity Overheads.* To measure the overhead of Memtrade, we run the YCSB workload over Redis with 50% remote memory. Integrity hashing increases remote memory access latency by 24.3% (22.9%) at the median (p99). Hashing and key replacement cause 0.9% memory overhead for the consumer. Due to fragmentation in the producer VM, the consumer needs to consume 16.7% extra memory over the total size of actual KV pairs. Note that this fragmentation overhead would be same if the consumer had to store the KVs in its local memory.

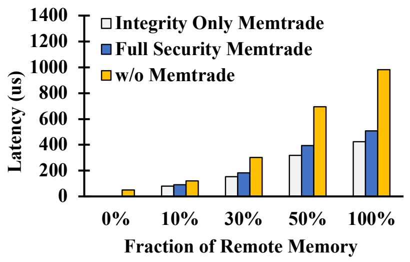


(a) **Placement**

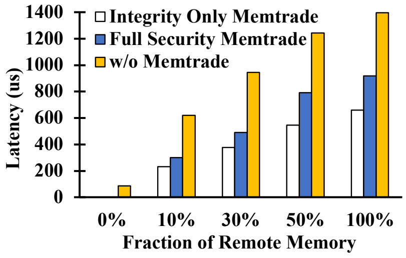


(b) **Cluster-wide Utilization**

Fig. 13. Simulation of remote memory usage shows the broker allocates most requests and improves cluster-wide utilization.



(a) **Average Latency**



(b) **p99 Latency**

Fig. 14. Benefit of Memtrade with various configurations. Without Memtrade, remote requests are served from SSD.

Encryption and key substitution increase latency by another 19.8% (14.7%) at the median (p99). Due to padding, encryption increases the memory overhead by another 25.2%. Fragmentation in the producer VM causes 6.1% memory overhead. Note that for trusted producers, or for non-sensitive consumer data, encryption can be disabled.

*Application-Level Performance.* We run YCSB over Redis with different consumer VM configurations and security modes. The consumer memory size is configured such that Redis needs at least x% ($x \in \{0, 10, 30, 50, 100\}$) of its working set to be in remote Memtrade memory. If remote memory is not available, the I/O operation is performed using SSD.

For the 0% configuration, the entire workload fits in the consumer VM's memory, and is fully served by its local Redis. Figure 14 shows that an application benefits from using remote memory as additional cache instead of missing to disk. For the fully-secure KV cache interface, Memtrade improves average latency by 1.3–1.9×, and p99 latency by 1.5–2.1×. In the case of non-sensitive data with only integrity checks, Memtrade provides 1.45–2.3× and 2.1–2.7× better average and p99 latencies, respectively.

We also implemented a transparent swap-based consumer interface, built on top of Infiniswap [74]. When consuming remote memory via fully-secure remote paging, Memtrade's performance drops due to hypervisor swapping overhead – average and p99 latency drop by 0.95–2.1× and 1.1–3.9×, respectively. However, given a faster swapping mechanism [95, 103] or faster network (e.g., RDMA), Memtrade swap is likely to provide a performance benefit to consumers.

*Cluster Deployment.* To measure the end-to-end benefit of Memtrade, we run 110 applications on CloudLab – 64 producer and 46 consumer VMs, randomly distributed across the cluster. Producers run six workloads described earlier, while consumers run YCSB on Redis, configured with 10%, 30%, and 50% remote memory. The total consumer and producer memory footprints are 0.7 TB and 1.3 TB, respectively; our cluster has a total of 2.6 TB. Memtrade benefits the consumers even in a cluster setting (Table 2). Memtrade improves the average latency of consumer applications by 1.6–2.8×, while degrading the average producer latency by 0.0–2.1%.

## 8 DISCUSSION

*Network Stack.* As public cloud environments are typically virtualized, inter-VM networking incurs additional latency due to traversing both kernel and hypervisor networking stacks. As shown in Figure 15, kernel-bypass networking (e.g., DPDK, RDMA) offers lower latency in virtualized environments; with RDMA and SR-IOV, the additional overhead is minimal compared to bare-metal machines. Unfortunately, RDMA is not widely available in existing public clouds. While DPDK is slower than RDMA, it still achieves single-digit $\mu s$ latencies for small objects. However, it consumes an excessive amount of CPU due to polling (nearly 100% of local CPU for larger objects), which would require producers to have extremely idle CPU resources to be practical.

CXL [18] is a new processor-to-peripheral/accelerator cache-coherent interconnect protocol that builds on and extends the existing PCIe protocol by allowing coherent communication between the connected devices.[2] It provides byte-addressable memory in the same physical address space and allows transparent memory allocation using standard memory allocation APIs. It also allows cache-line granularity access to the connected devices and underlying hardware maintains cache-coherency and consistency. With PCIe 5.0, CPU-to-CXL interconnect bandwidth is similar to the cross-socket interconnects on a dual-socket machine [124]. CXL-Memory access latency is also similar to the NUMA access latency. CXL adds around 50-100 nanoseconds of extra latency over normal DRAM access.

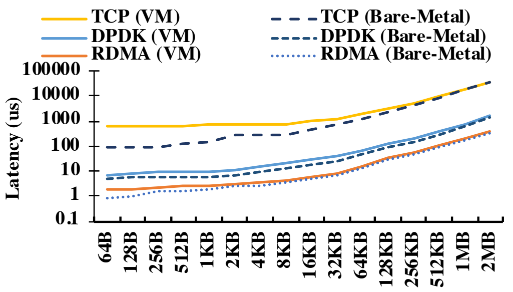


Fig. 15. Network latency using VMs vs. bare-metal hosts for various protocols and different object sizes.

Because of its universal availability, Memtrade currently uses TCP networking, despite its relatively high overhead and other limitations. For deployments in private clouds, replacing TCP with RDMA or even CXL-enabled rack-servers would improve performance significantly.

*Consumer Abstractions.* Memtrade is designed considering the readily deployability feature for most of the common usages in existing cloud infrastructure. To hold its ubiquitousness, we design the consumer APIs thinking TCPs as the most common network medium. Although it needs some re-write or modification to the consumer applications, on TCP networks, caching applications can usually sustain the network communication overhead. Swapping to remote memory can be a viable and transparent abstraction for consumer applications [56, 88, 95]. However, to maintain the performance while remote swapping, we need to ensure the network stack is at least as performant as RDMA over Infiniband. With faster and next-generation cache-coherent CXL interconnects, we are confident on Memtrade's applicability. In such a setup, we can design more transparent and

[2]Prior industry standards in this space such as CCIX [14], OpenCAPI [31], Gen-Z [23] etc. have all come together under the banner of CXL consortium. While there are some related research proposals (e.g., [87]), CXL is the de facto industry standard at the time of writing this paper.

| Producer Application | Avg. Latency (ms) w/o Harvester | Avg. Latency (ms) w/ Harvester | Consumer Application | Avg. Latency (ms) w/o Memtrade | Avg. Latency (ms) w/ Memtrade |
|---|---|---|---|---|---|
| Redis | 0.08 | 0.08 | Redis 0% | 0.62 | – |
| Memcached | 0.82 | 0.83 | Redis 10% | 1.10 | 0.71 |
| MySQL | 1.57 | 1.60 | Redis 30% | 1.54 | 0.88 |
| Storm | 5.33 | 5.47 | Redis 50% | 2.49 | 0.89 |

Table 2. Memtrade benefits consumers at a small cost to producers.

performant interfaces to the consumers to consume remote memory efficiently. OS level features (i.e., migration-based page placement [96]) can be utilized to design new consumer APIs.

## 9 RELATED WORK

*Remote Memory.* Existing work on *remote-memory-enabled* datacenter [47, 48, 51, 71, 74, 75, 79, 82, 85, 91, 99, 103, 108] assume that *memory is contained within the same organization and shared among multiple cooperative applications.* Given the large amount of idle memory and diverse consumer applications and workloads, public clouds serve as a promising environment to exploit remote memory.

*Public Cloud Spot Marketplaces.* Amazon AWS, Microsoft Azure, and Google Cloud offer spot instances [35] – a marketplace for unutilized public cloud VMs that have not been reserved, but have been provisioned by the cloud provider. AWS allows customers to bid on spot instances while Azure and Google Cloud [32, 38] sell them at globally fixed prices. Prior work explored the economic incentives for spot instances [45, 46, 54, 70, 73, 111, 112, 118, 121, 123, 125]. However, they used full spot instances to produce memory; Memtrade is more generic, enabling fine-grained consumption of excess memory from any instance type.

*Single-Server Memory Management.* Reclaiming idle memory from applications has been explored in many different contexts, e.g., physical memory allocation across processes by an OS [44, 63, 66], ballooning in hypervisors [106, 110, 116], transcendent memory for caching disk swap [92–94], etc. Our harvesting approach is related to application-level ballooning [105]. However, in most prior work, applications belong to the same organization while Memtrade harvests and grants memory across tenants. Multi-tenant memory sharing [46, 62, 70] has considered only single-server settings, limiting datacenter-wide adoption.

*Resource Autoscaling and Instance Rightsizing.* Cloud orchestration frameworks and schedulers [7, 28, 49, 107, 113] can automatically increase or decrease the number of instances based on the task arrival rate. However, users still need to statically determine the instance size before launching a task, which may lead to resource overprovisioning.

Instance rightsizing [39, 50, 52, 104, 119] automatically determines the instance size based on a task's resource consumption. In existing solutions, cloud providers are fully responsible for resource reclamation and performance variability in producer VMs. Memtrade is by design more conservative: producers can generate and reclaim memory capacity at any time. Even with rightsizing, servers may have idle memory that can be harvested and offered to consumer VMs.

*Memory Harvesting.* Harvesting unused resources of a traditional server or VM is a concern in many hyperscale datacenters [69, 109]. Earlier research mostly focus on the unallocated memory of a VM as the harvesting target. However, besides that unallocated memory, there are significant amount of memory that has been allocated to an application; the application used that for a while; and then, remained unutilized for a significant amount of time [85, 89, 96, 120]. Memtrade harvester considers all kind of idle memory.

## 10 CONCLUDING REMARKS

We present Memtrade, a readily deployable system for the realization of disaggregated memory marketplace on existing clouds. With the rising popularity of serverless and disaggregated storage, there will be increased demand for offering disaggregated computing resources in public clouds, and our work attempts to apply a similar approach to memory. This opens several interesting research directions for future work, including exploring whether other resources, such as CPU and persistent memory, can be offered in a similar manner via a disaggregated-resource market.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for insightful feedback that helped improve the paper. This work was supported in part by National Science Foundation grants (CNS-1845853, CNS-2104243, CNS-2143868) and gifts from VMware and Meta.

## REFERENCES

[1] Alibaba Cluster Trace 2018. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.
[2] Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless/.
[3] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/.
[4] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.
[5] Amazon Elastic Block Store. https://aws.amazon.com/ebs/.
[6] Amazon ElastiCache. https://aws.amazon.com/elasticache/.
[7] Apache Hadoop NextGen MapReduce (YARN). http://goo.gl/etTGA.
[8] AWS Lambda. https://aws.amazon.com/lambda/.
[9] AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html.
[10] AWS Spot Instance Pricing History. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances-history.html.
[11] AWS Transit Gateway. https://aws.amazon.com/transit-gateway.
[12] Azure Cache for Redis. https://azure.microsoft.com/en-us/services/cache/.
[13] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[14] CCIX. https://www.ccixconsortium.com/.
[15] Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.
[16] Classification for Biospecies 3. https://www.kaggle.com/olgabelitskaya/tf-cats-vs-dogs/version/2.
[17] CloudLab. https://www.cloudlab.us/.
[18] Compute Express Link (CXL). https://www.computeexpresslink.org/.
[19] Custom Machine Types. https://cloud.google.com/custom-machine-types/.
[20] etcd. https://github.com/etcd-io/etcd.
[21] Facebook and Amazon are causing a memory shortage. https://www.networkworld.com/article/3247775/facebook-and-amazon-are-causing-a-memory-shortage.html.
[22] Frontswap. https://www.kernel.org/doc/html/latest/vm/frontswap.html.
[23] Gen-Z. https://genzconsortium.org/.
[24] Global VNet Peering now generally available? https://azure.microsoft.com/en-us/blog/global-vnet-peering-now-generally-available/.
[25] Google Cluster Trace 2019. https://github.com/google/cluster-data/blob/master/ClusterData2019.md.
[26] Graph Analytics Benchmark in CloudSuite. http://parsa.epfl.ch/cloudsuite/graph.html.
[27] Idle Memory Tracking. https://www.kernel.org/doc/Documentation/vm/idle_page_tracking.txt.
[28] Kubernetes. http://kubernetes.io.
[29] Linux Virtual Machines Pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/.
[30] MemCachier. https://www.memcachier.com/.
[31] OpenCAPI. https://opencapi.org/.
[32] Preemptible VM Instances. https://cloud.google.com/compute/docs/instances/preemptible.
[33] Redis, an in-memory data structure store. http://redis.io.
[34] Redis Labs. https://redislabs.com/.
[35] Spot Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html.
[36] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net.
[37] Tracking Pressure-Stall Information. https://lwn.net/Articles/759781/.

[38] Use low-priority VMs with Batch. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms.

[39] Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

[40] VM instances pricing. https://cloud.google.com/compute/vm-instance-pricing.

[41] What is VPC peering? https://docs.aws.amazon.com/vpc/latest/peering/what-is-vpc-peering.html.

[42] Who's using Redis? https://redis.io/topics/whos-using-redis.

[43] zram. https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[44] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. *SIGPLAN*, 2017.

[45] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation*, 2013.

[46] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem. Ginseng: Market-driven memory allocation. *SIGPLAN*, 2014.

[47] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *USENIX ATC*, 2018.

[48] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SoCC*, 2017.

[49] S. Alamro, M. Xu, T. Lan, and S. Subramaniam. CRED: Cloud right-sizing to meet execution deadlines and data locality. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016.

[50] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[51] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In *EuroSys*, 2020.

[52] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini. Providing SLOs for resource-harvesting VMs in cloud platforms. In *OSDI*, 2020.

[53] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[54] O. A. Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. The Resource-as-a-Service (RaaS) cloud. In *HotCloud*, 2012.

[55] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *OSDI*, 2020.

[56] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, 2021.

[57] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *FAST*, 2020.

[58] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *SIGKDD*, 2016.

[59] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *IPDPSW*, 2016.

[60] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *HotCloud*, 2015.

[61] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *NSDI*, 2016.

[62] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *USENIX ATC*, 2017.

[63] E. G. Coffman, Jr. and P. J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[64] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[65] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.

[66] P. J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[67] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM footprint with NVM in Facebook. In *EuroSys*, 2018.

[68] A. Eisenman, M. Naumov, D. Gardner, M. Smelyanskiy, S. Pupyrev, K. Hazelwood, A. Cidon, and S. Katti. Bandana: Using non-volatile memory for storing deep learning models. *SysML*, 2019.

[69] A. Fuerst, S. Novaković, I. n. Goiri, G. I. Chaudhry, P. Sharma, K. Arya, K. Broas, E. Bak, M. Iyigun, and R. Bianchini. Memory-harvesting vms in cloud platforms. In *ASPLOS*, 2022.

[70] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. Ginseng: Market-driven LLC allocation. In *USENIX ATC*, 2016.

[71] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[72] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu. When cloud storage meets RDMA. In *NSDI*, 2021.

[73] A. Ghalayini, J. Geng, V. Sachidananda, V. Sriram, Y. Geng, B. Prabhakar, M. Rosenblum, and A. Sivaraman. CloudEx: A fair-access financial exchange in the cloud. In *HotOS*, 2021.

[74] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[75] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *HotNets*, 2013.

[76] S. L. Ho and M. Xie. The use of ARIMA models for reliability forecasting and analysis. *Comput. Ind. Eng.*, 1998.

[77] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang. Kinetic modeling of data eviction in cache. In *USENIX ATC*, 2016.

[78] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, 2010.

[79] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *DATE*, 2016.

[80] S. Keshav and S. Kesahv. *An engineering approach to computer networking: ATM networks, the Internet, and the telephone network*, volume 116. Addison-Wesley Reading, 1997.

[81] A. Klimovic and C. Kozyrakis. ReFlex : Remote Flash = Local Flash. In *ASPLOS*, 2017.

[82] A. Klimovic, C. Kozyrakis, and B. John. Flash Storage Disaggregation. In *EuroSys*, 2016.

[83] A. Klimovic, H. Litz, and C. Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *USENIX ATC*, 2018.

[84] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[85] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, 2019.

[86] C. Lee and J. Ousterhout. Granular computing. In *HotOS*, 2019.

[87] S.-s. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee. MIND: In-network memory management for disaggregated data centers. In *SOSP*, 2021.

[88] Y. Lee, H. A. Maruf, M. Chowdhury, A. Cidon, and K. G. Shin. Hydra : Resilient and highly available remote memory. In *FAST*, 2022.

[89] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. *arXiv preprint arXiv:2203.00241*, 2022.

[90] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Cluster Computing*, 2005.

[91] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH*, 2009.

[92] D. Magenheimer. Transcendent memory on Xen. *Xen Summit*, 2009.

[93] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Paravirtualized paging. In *WIOV*, 2008.

[94] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Transcendent memory and linux. In *Proceedings of the Linux Symposium*, pages 191–200. Citeseer, 2009.

[95] H. A. Maruf and M. Chowdhury. Effectively Prefetching Remote Memory with Leap. In *USENIX ATC*, 2020.

[96] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan. TPP: Transparent page placement for CXL-enabled tiered memory, 2022.

[97] I. Müller, R. F. Bruno, A. Klimovic, G. Alonso, J. Wilkes, and E. Sedlar. Serverless clusters: The missing piece for interactive batch applications? In *SPMA*, 2020.

[98] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at Facebook. In *NSDI*, 2013.

[99] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *EuroSys*, 2018.

[100] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[101] K. Psounis, B. Prabhakar, and D. Engler. A randomized cache replacement scheme approximating LRU. In *Proceedings of the 34th Annual Conference on Information Sciences and Systems*, March 2000.

[102] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[103] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*, 2020.

[104] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes. Autopilot: Workload autoscaling at Google. In *EuroSys*, 2020.

[105] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone. Application level ballooning for efficient server consolidation. In *EuroSys*, 2013.

[106] J. H. Schopp, K. Fraser, and M. J. Silbermann. Resizing memory with balloons and hotplug. In *Proceedings of the Linux Symposium*, volume 2, pages 313–319, 2006.

[107] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[108] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[109] P. Sharma, A. Ali-Eldin, and P. Shenoy. Resource deflation: A new approach for transient resource reclamation. In *EuroSys*, 2019.

[110] P. Sharma, P. Kulkarni, and P. Shenoy. Per-VM page cache partitioning for cloud computing platforms. In *COMSNETS*, 2016.

[111] S. Shastri, A. Rizk, and D. Irwin. Transient guarantees: Maximizing the value of idle cloud capacity. In *SC*, 2016.

[112] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. In *2012 Proceedings IEEE Infocom*, pages 190–198. IEEE, 2012.

[113] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with Borg. In *EuroSys*, 2015.

[114] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.

[115] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache modeling and optimization using miniature simulations. In *USENIX ATC*, 2017.

[116] C. A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI*, 2002.

[117] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient MRC construction with SHARDS. In *FAST*, 2015.

[118] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *EuroSys*, 2017.

[119] Y. Wang, K. Arya, M. Kogias, M. Vanga, A. Bhandari, N. J. Yadwadkar, S. Sen, S. Elnikety, C. Kozyrakis, and R. Bianchini. SmartHarvest: Harvesting idle CPUs safely and efficiently in the cloud. In *EuroSys*, 2021.

[120] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos. TMO: Transparent memory offloading in datacenters. In *ASPLOS*, 2022.

[121] H. Xu and B. Li. Dynamic cloud pricing for revenue maximization. *IEEE Transactions on Cloud Computing*, 1(2):158–171, 2013.

[122] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *OSDI*, 2020.

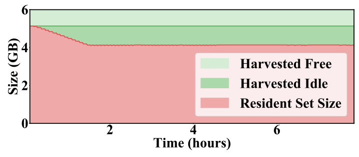
[123] Q. Zhang, Q. Zhu, and R. Boutaba. Dynamic resource allocation for spot markets in cloud computing environments. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 178–185. IEEE, 2011.

[124] W. Zhao and J. Ning. Project Tioga Pass Rev 0.30 : Facebook Server Intel Motherboard V4.0 Spec. https://www.opencompute.org/documents/facebook-server-intel-motherboard-v40-spec.

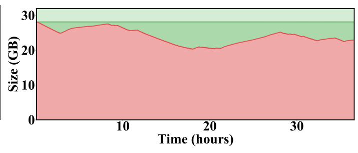
[125] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang. How to bid the cloud. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 71–84. ACM, 2015.

[126] Q. Zheng, K. Ren, G. Gibson, B. Settlemyer, and G. Grider. DeltaFS: exascale file systems scale better without dedicated servers. In *PDSW*, 2015.

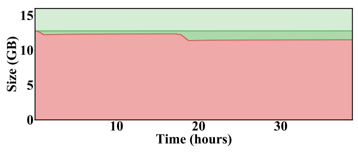
## A APPENDIX



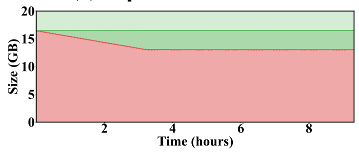
(a) **Zipfian on Redis**



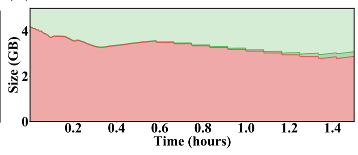
(b) **MemCachier on memcached**



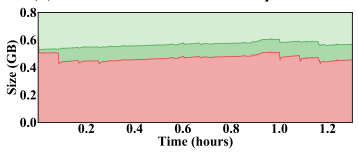
(c) **MemCachier on MySQL**



(d) **Image classification on XG-Boost**



(e) **Yahoo Streaming on Apache Storm**



(f) **CloudSuite Web Serving**

Fig. 16. Unallocated represents the part of memory not allocated to the application; harvested means the portion of application's memory which has been swapped to disk; Silo denotes the part of memory used by Silo to buffer reclaimed pages; RSS consists of application's anonymous pages, mapped files, and page cache that are collected from the cgroup's stats file.

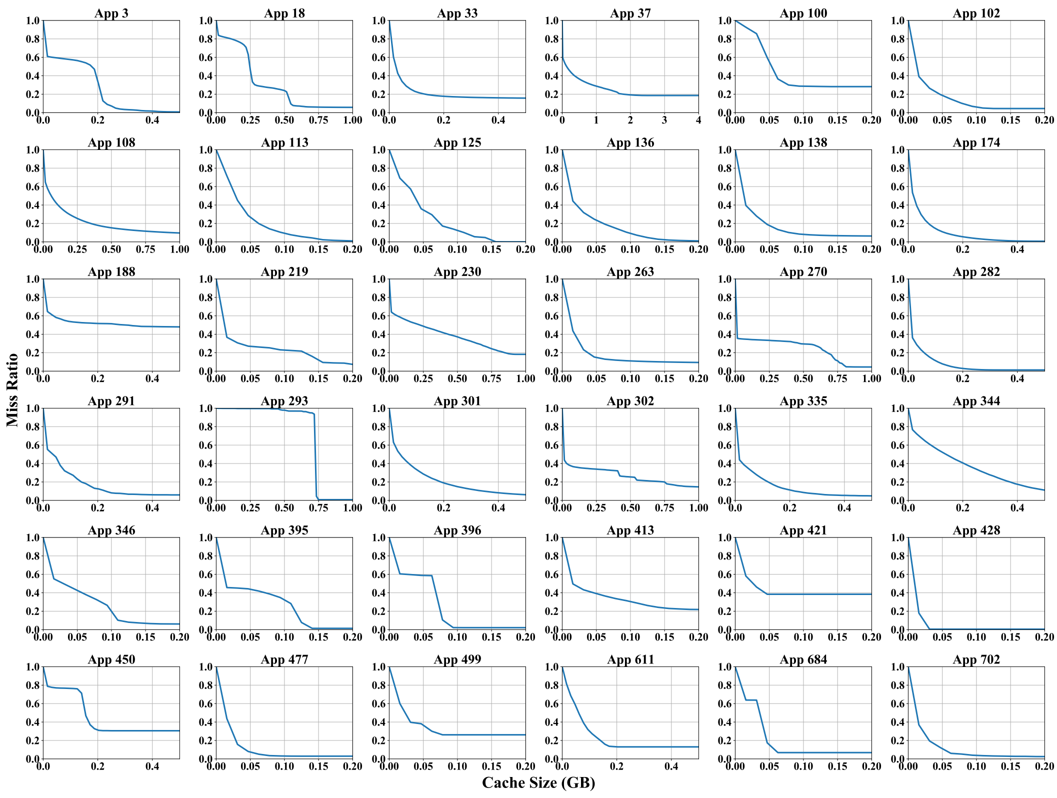


Fig. 17. Miss Ratio Curves of 36 MemCachier Applications

Received February 2023; revised March 2023; accepted April 2023