Memory-Harvesting VMs in Cloud Platforms

Alexander Fuerst[1]
Indiana University

Stanko Novaković
Microsoft Research

Íñigo Goiri
Microsoft Research

Gohar Irfan Chaudhry
Microsoft Research

Prateek Sharma
Indiana University

Kapil Arya
Microsoft Research

Kevin Broas
Microsoft Azure

Eugene Bak
Microsoft Azure

Mehmet Iyigun
Microsoft Azure

Ricardo Bianchini
Microsoft Research

ABSTRACT

Cloud platforms monetize their spare capacity by renting "Spot" virtual machines (VMs) that can be evicted in favor of higher-priority VMs. Recent work has shown that resource-harvesting VMs are more effective at exploiting spare capacity than Spot VMs, while also reducing the number of evictions. However, the prior work focused on harvesting CPU cores while keeping memory size fixed. This wastes a substantial monetization opportunity and may even limit the ability of harvesting VMs to leverage spare cores. Thus, in this paper, we explore memory harvesting and its challenges in real cloud platforms, namely its impact on VM creation time, NUMA spanning, and page fragmentation. We start by characterizing the amount and dynamics of the spare memory in Azure. We then design and implement memory-harvesting VMs (MHVMs), introducing new techniques for memory buffering, batching, and pre-reclamation. To demonstrate the use of MHVMs, we also extend a popular cluster scheduling framework (Hadoop) and a FaaS platform to adapt to them. Our main results show that (1) there is plenty of scope for memory harvesting in real platforms; (2) MHVMs are effective at mitigating the negative impacts of harvesting; and (3) our extensions of Hadoop and FaaS successfully hide the MHVMs' varying memory size from the users' data-processing jobs and functions. We conclude that memory harvesting has great potential for practical deployment and users can save up to 93% of their costs when running workloads on MHVMs.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Cloud computing, memory management, resource harvesting.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507725

ACM Reference Format:
Alexander Fuerst[1], Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-Harvesting VMs in Cloud Platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3503222.3507725

1 INTRODUCTION

Motivation. Cloud platforms such as AWS, Azure, and GCP must leave some spare capacity (*e.g.*, CPU cores, memory, network bandwidth) to ensure that the customers' workloads can scale out and experience high platform availability [4]. At the same time, the platforms monetize this unallocated capacity by offering lower-priority evictable VMs, often called Spot VMs, at discounted prices [3, 6, 10]. If the platform needs the resources taken by a Spot VM for a regular-priority (or simply regular) VM, it evicts the Spot VM to free up its resources. Due to their characteristics, customers often use Spot VMs for batch jobs or other workloads that can tolerate evictions.

Unfortunately, Spot VMs receive a fixed amount of physical resources and (1) cannot shrink to prevent evictions or (2) grow to use any additional unallocated resources that might become available on the same servers. To address these limitations, prior work has proposed dynamic resource harvesting by creating evictable VMs that can grow and shrink based on the amount of available resources in their host servers [4, 43, 49]. For example, a Harvest VM grows when it lands on a server with more unallocated resources than its minimum size or when a regular VM departs from the server, and shrinks when a new regular VM lands on the server [4].

These prior works have focused mainly on harvesting cores, leaving unallocated memory on the table. Memory is expensive (*e.g.*, roughly 50% of server costs today [39]), so failing to monetize some of it represents a significant loss for the cloud provider. Moreover, harvesting cores only either (1) requires a large enough memory for the maximum number of cores that can be harvested or (2) restricts the use of the harvesting VMs to CPU-bound workloads with small memory footprints.

Our work. In this paper, we extend the work of Ambati *et al.* on Harvest VMs [4] to also harvest unallocated memory. We refer to our extension as memory-harvesting VMs (MHVMs). Each MHVM

[1]Fuerst (alfuerst@iu.edu) was an intern at Microsoft Research.

has a minimum memory size, but receives more memory beyond this minimum depending on the amount of unallocated memory in its host server. When a regular VM departs the server, the MHVM receives the newly unallocated memory and expands its allocation. When a new regular VM arrives, the MHVM reduces its memory allocation to make room for the VM. The MHVM is only evicted when the platform needs its minimum set of resources for a new regular VM. The cloud provider can hide the complexity of dynamic resource changes from customers by using MHVMs to create cheap SaaS (Software-as-a-Service), PaaS (Platform-as-a-Service), and FaaS (Function-as-a-Service) offerings.

While memory harvesting is highly desirable, it introduces multiple challenges that do not exist in core harvesting. First, when a regular VM arrives, memory harvesting entails de-allocating a large number of pages in the MHVM and returning them to the hypervisor. This process can take tens of seconds and lengthen the regular VM creation time. Second, because MHVMs may harvest memory from multiple NUMA nodes, they may cause incoming regular VMs to span NUMA nodes and potentially suffer performance loss. Third, the de-allocation/re-allocation process may cause large pages (*e.g.*, 2 MB) to be broken down into smaller non-contiguous 4 KB pages, again potentially hurting the performance of regular VMs. Though the second and third challenges may also imply performance issues for the workloads running on MHVMs, they are not as serious because these workloads should be inherently amenable to performance variability (including evictions). The prior work that studied memory harvesting in virtualized environments [14, 43] considered *none* of these challenges and performance implications.

To understand and address these challenges, we first quantify the opportunity for harvesting unallocated memory in Azure. In particular, we study how much and often MHVMs would have to be resized, and the corresponding impact on regular VMs (*i.e.*, VM creation time, NUMA spanning, and large page fragmentation).

Based on this characterization, we design memory harvesting managed by a server-local agent. We leverage memory hot-add and hot-remove mechanisms that exist in modern operating systems, such as Linux and Windows, to grow and shrink VMs. We minimize the impact on VM creation time by first reducing the time to reclaim memory from MHVMs using batching and pre-reclamation. We also introduce an unallocated memory *buffer* that hides most of the reclamation time. This buffer also reduces the likelihood of NUMA spanning. Our agent handles this buffer asynchronously and evicts the MHVM if it is not returning enough memory or generating too much NUMA spanning. The agent also exposes the NUMA topology to MHVMs and resizes memory in a NUMA-aware fashion. To avoid large page fragmentation, it reclaims memory in 2 MB contiguous chunks.

To demonstrate the use of our MHVMs in a way that is transparent to customers workloads, we then modify a cluster scheduling framework (Hadoop) and a FaaS platform that emulates Azure Functions to adapt to memory resizing. Our memory-harvesting Hadoop (or simply MH-Hadoop) and MH-FaaS increase the performance at which we can reclaim memory. Moreover, they reclaim large contiguous chunks of memory which eliminates fragmentation and increases the reclamation throughput.

Finally, we experimentally evaluate MHVMs, MH-Hadoop, and MH-FaaS using both microbenchmarks and representative workloads from Azure Functions. Our results show that we can harvest memory without impacting the existing workloads running in regular VMs and reduce the costs for the frameworks running in the harvested resources by 92.8%.

Summary. Our main contributions are:

- A characterization of the amount and dynamics of unallocated memory in a real cloud platform;
- The design and implementation of MHVMs, including novel techniques for memory buffering, batching, and pre-reclamation;
- Extensions of Hadoop and a FaaS platform to adapt to changes in the amount of harvested memory; and
- Extensive results from microbenchmarks and real cloud workloads exploring the impact of memory harvesting on NUMA spanning, page fragmentation, and VM creation time.

2 BACKGROUND AND RELATED WORK

2.1 Cloud platforms and spare resources

VM deployment. Cloud platforms deploy customer VMs across multiple geographical regions, each comprising multiple server clusters. Within a region, a bin-packing scheduler assigns VMs to servers. An agent running on each server is responsible for managing VMs through their lifecycles (*e.g.*, VM creation). Provisioning resources for each new VM takes many seconds [26], so platforms try to optimize it (*e.g.*, [40]) for improved customer experience.

Spare/unallocated cores, Spot, and Harvest VMs. As we illustrate in the next section, platforms often end up with spare resources around its clusters that are not allocated to any customers. Fixed-size Spot VMs are the current approach platforms use to monetize this spare capacity. However, Ambati *et al.* proposed a better approach to leveraging spare resources: Harvest VMs [4]. These VMs receive a fixed amount of memory, but can harvest any cores that happen to be unallocated on the same server. Harvest VMs produce fewer evictions and lower costs for customers than Spot VMs. Our work extends the concept of Harvest VMs to memory, while solving the challenges of memory harvesting without impacting workload performance and customer satisfaction.

Other variable-resource VMs. Burstable VMs [5, 9] are similar to Harvest VMs in that they dynamically adjust the number of physical cores assigned to a VM. However, the adjustment is based on the concept of "credit" and not on whether there are unallocated cores. Like Harvest VMs, the memory size is fixed.

Elastic VMs [49] build upon Harvest VMs to harvest both unallocated cores and allocated but temporarily idle cores. Again, memory size is fixed. Sharma *et al.* proposed Deflatable VMs [43], a multi-level resource reclamation approach for explicitly adapting applications, OSes, and hypervisors to any available (unallocated or simply unused) resources. They considered none of the challenges we address in this paper.

Other approaches to harvesting. Prior works have considered co-locating batch workloads with latency-sensitive services on bare-metal servers [21, 24, 45, 46, 51, 57, 59]. In contrast, we target platforms with virtualized infrastructure, opaque VMs, and strict performance requirements. Thus, these works considered none of the challenges we address.

2.2 Hypervisor-level memory management

Mechanisms and policies. Modern hypervisors support different memory reclamation mechanisms, each with their own tradeoffs. *Ballooning* [48] uses a kernel driver inside the guest to allocate guest physical memory pages, and notify the host that the corresponding host physical pages can be assigned to another guest. Allocating pages involves making calls to the guest's memory allocator and other guest OS memory management overheads that may severely limit reclamation throughput. On top of ballooning, the guest may *hot-remove* the allocated pages from the guest physical address space. Similarly, most operating systems and hypervisors today support extending the guest physical address space at run time via *hot-add* [23, 38]. Our work leverages these mechanisms.

Policies for dynamic VM memory sizing face the fundamental size vs performance tradeoff, using techniques such as working set estimation [8, 60], and application and OS performance metrics [16, 19, 50, 56]. Our work focuses on memory harvesting in production public cloud platforms, where such techniques are considered invasive and cannot be universally and safely applied. We also focus on Harvest VMs, which run custom frameworks that are aware of resource dynamicity and can shrink their working sets on-demand or pro-actively.

Allocation to NUMA nodes. Modern servers in datacenters usually have multiple Non-Uniform Memory Access (NUMA) nodes [44]. When the agent in the server creates a VM, it tries to assign all the memory from a single (local) NUMA node for low memory access latency. If there is not enough memory to allocate the VM in the required NUMA node, the hypervisor will assign some memory from a different (remote) NUMA node [25] – we refer to this as NUMA spanning. Accessing a remote NUMA node has higher latency (*e.g.*, roughly 150 nanoseconds in Intel servers) as the request needs to cross the interconnect between the nodes. In some cases, the hypervisor will surface the underlying NUMA topology to the VM (*i.e.*, vNUMA), in which case the guest OS can optimize for locality of access. However, individual vNUMA nodes can still span the physical NUMA nodes.

Using large pages. To reduce TLB pressure and shorten page walks, large pages (*i.e.*, 2 MB pages on x86 instead of 4 KB) are commonly used and especially in virtualized environments in the hypervisor-level page tables. Some operating systems even include support for huge pages (*e.g.*, 1 GB) [2]. Large pages produce significant performance improvements for many workloads, as we illustrate later in Section 4.2 using representative cloud workloads.

Summary. Prior work [4, 14, 43] did not consider the three major issues for production deployment of memory harvesting in real clouds: (1) The impact of memory harvesting on VM creation performance, (2) host memory fragmentation, and (3) NUMA spanning. Our work develops new techniques to address these challenges and improves existing techniques, such as the speed of runtime memory resizing, and develop new ones to mitigate these issues.

3 MEMORY USAGE IN AZURE

In this section, we characterize the resource allocation and its dynamics in Azure. We analyze data from thousands of (randomly-selected) servers from 6 geographical regions over 3 weeks in 2021. The data spans 4 server generations with 7 different configurations.

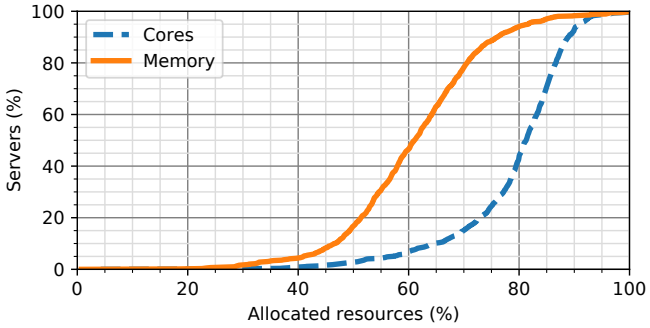


Figure 1: Allocated resources per server in increasing order. There is room for harvesting both cores and memory.

Resource allocation. Figure 1 shows the percentage of memory allocated per server (orange curve) in our sample, in increasing order. For comparison, the figure also shows the percentage of cores allocated per server (blue curve), again in increasing order. More than half of the servers have an allocation of cores above 80%, while memory allocation is only between 50% and 70% for most servers. This shows that there is an opportunity to harvest both cores and memory. These findings are consistent with the ones in [12, 55].

Frequency of allocation changes. The amount of allocated resources changes as VMs arrive and depart at each server, so we now study the frequency of those events. Figure 2 shows CDFs of how many VM arrivals (orange curve) and departures (green curve) a server experiences per hour over all server hours (*i.e.*, #servers × #hours_in_3_weeks) in our dataset. The orange curve shows a CDF of the VM events (arrivals or departures) per hour. For example, in 92% of the hours, the number of events per hour is at most 2.

Interestingly, in 82% of the hours, servers see no VM arrivals or departures despite the high VM event traffic in the 6 regions. There are 4 main reasons for this high number. First, bin packing of VMs to servers tends to repeatedly target servers that provide the best fit; a VM that departs is quickly replaced by another VM. Second, long-lived VMs consume the vast majority of resources, meaning that some servers become full with those VMs and see no more events. Third, many servers host (typically long-lived) full-server VMs that leave no room for other VMs. Fourth, VM events typically decrease over nights and weekends to a much lower level than the datacenters' peak capacity. At the other end of the spectrum, we see a small percentage of hours with tens of events per hour. More importantly, there are no arrivals in 86% of the hours and less than 4 arrivals per hour in 96% of the hours. These results show that memory harvesting would need to reclaim memory infrequently.

Size of allocation changes. Figure 3 show the CDFs of how much the core (left) and memory (right) allocations change per event. Positive values represent VM arrivals and negative ones represent VM departures. As one would expect, the allocation increases and decreases are fairly symmetrical. Only 10% of allocation changes are of more than 8 cores and more than 32 GB of memory. More importantly, a large percentage of VM arrivals involve fairly small amounts of memory. For example, 60% of VM arrivals involve 16 GB or less. These results suggest that most memory reclamation operations would involve relatively small amounts of memory.

Amount of NUMA spanning. We quantify how much NUMA spanning VMs experience in Azure. We find that 0.9% of allocated

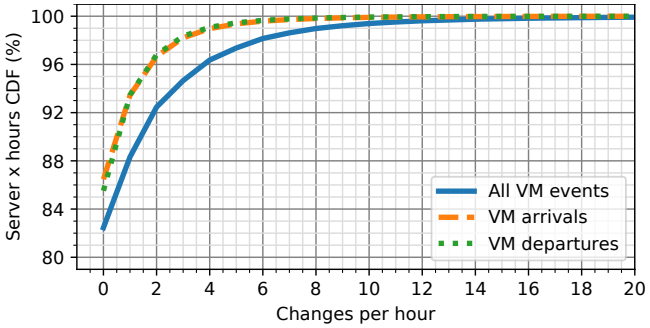


Figure 2: Resource changes per hour at each server over all hours. 82.5% of the hours involve no VM events, and in 86.5% of them there are no VM arrivals.

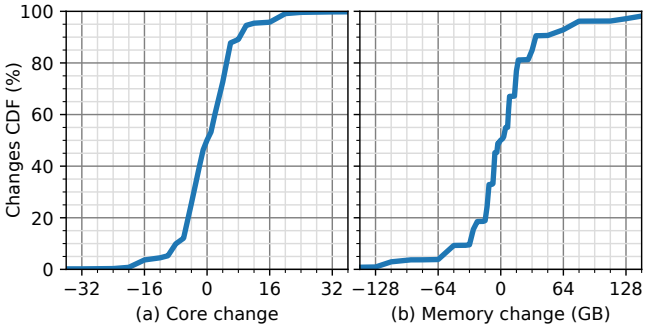


Figure 3: Core and memory allocation changes. Positive values mean VM arrivals. 90% of the VM creations involve 8 cores or fewer and 32 GB of memory or less.

memory is in a different NUMA node than the cores, and that only 2% of VMs experience spanning. These results show that customers experience spanning rarely, if at all, so memory harvesting must keep spanning down to avoid potential customer dissatisfaction.

Large page fragmentation. Finally, VMs currently do not experience any hypervisor-level large page fragmentation, because Azure allocates all of each VM's memory contiguously at its creation time. Again, this suggests that memory harvesting must keep fragmentation to a minimum.

Summary and implications. Our analysis of production data shows that there are substantial opportunities to harvest spare/unallocated memory in real platforms. It also suggests that the most challenging operation, memory reclamation from MHVMs, occurs tens of minutes apart the vast majority of the time. Moreover, each reclamation would involve less than 32 GB 90% of the time. Finally, the data suggests that harvesting must properly manage NUMA placement and hypervisor-level paging, since VMs currently enjoy local NUMA accesses and large-page second-level accesses.

4 MEMORY-HARVESTING VMS

As we show in Section 3, there is plenty of opportunity for harvesting unallocated memory. Thus, in this section, we first overview our design for Memory-Harvesting VMs (MHVMs) to monetize that memory. Then, we discuss the key challenges in implementing MHVMs and our techniques for preventing MHVMs from harming the performance of co-located VMs.

4.1 Overview

MHVMs are a new type of evictable VM that builds upon the harvesting of cores in Harvest VMs [4]. Customers can deploy MHVMs to the platform just as any other VM type. A MHVM starts with a minimum memory allocation and dynamically grows to take unallocated memory (up to its maximum) when other VMs leave the server. It also shrinks when its memory is needed by a new VM arriving in the same server. The harvesting of cores happens exactly like in Harvest VMs.

Our design has three key aspects. First, for ease of production deployment, our design minimizes the number of changes needed by the cloud platform: only the agent that runs on the servers (Section 2.1) knows about the memory harvesting. Second, our modified agent leverages mechanisms from the hypervisor to effect changes to the MHVMs' memory size. Finally, we build a harvesting manager that runs on each MHVM to serve as an intermediary between the MHVM and the applications running on it.

Server agent. We modify the agent to continually adjust the memory assigned to the MHVMs, based on the amount of available memory in the server. Our agent also implements multiple techniques for preventing MHVMs from harming the performance of the co-located VMs (Section 4.3).

Harvesting manager. Our harvesting manager runs as a process on each MHVM. It monitors the changes in MHVM's memory size and notifies the applications running on it to adjust accordingly. Section 5 describes the application frameworks that we extend to adjust to MHVMs.

Resizing steps. MHVM resizing operations occur when a VM needs to be created on a server or has departed from it. In the former case, the server agent calls a hypervisor API to downsize each MHVM by the needed amount, the hypervisor then tells the resizing driver of the guest OS in each MHVM how much memory it has to return to the hypervisor. The driver typically complies successfully but may take a substantial amount of time to do so. Inside the MHVM, the harvesting manager either detects the change by monitoring the free space in the VM or by receiving a notification from the hypervisor (Section 4.4). Upon a VM departure, the interactions are similar but memory is added to the MHVMs.

Though the concept of MHVMs is fairly simple, implementing them in a way that does not harm the performance of co-located regular VMs is a major challenge.

4.2 Challenges in real cloud platforms

The challenges relate to the impact of memory harvesting on VM creation time, NUMA locality, and large pages.

VM creation times. Being able to quickly add, and especially remove, memory from MHVMs is critical to ensuring that cloud SLAs, such as VM creation time, are not adversely impacted. Under high server utilization, memory reclamation can be on the critical path before a new VM can be launched, and thus reclamation throughput becomes a primary consideration. For instance, reclamation throughput has been measured at 2-3 GB/s [18, 23], which translates to a shrinking time (and thus VM creation time) of tens of seconds. Public cloud providers are highly unlikely to accept a degradation of this magnitude in VM creation time due to harvesting.

Workload	Description	Metric	Memory
BA	Business analytics queries [32]	Runtime	18 GB
KV	Distributed key-value store [27]	99^{th} latency	32 GB
SQL	Relational database [33]	95^{th} latency	16 GB
Search	Serving index requests [31]	99^{th} latency	128 GB
Web	3-tier web application [11]	Throughput	4 GB
ML	Machine learning training [1]	Runtime	64 GB

Table 1: Popular internal and external workloads in Azure.

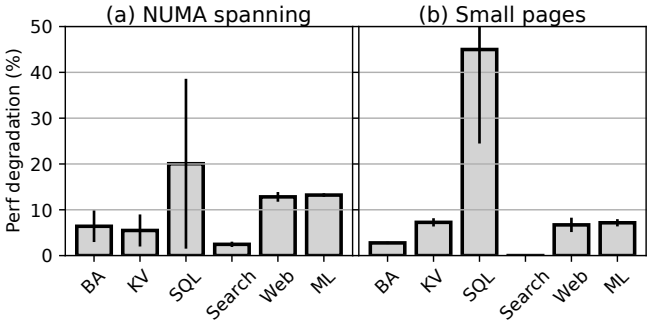


Figure 4: Potential impact of (a) NUMA spanning and (b) small pages on cloud workloads.

NUMA locality. Consider a scenario where the hypervisor grows the memory of a MHVM on one of the NUMA nodes. A new VM that arrives later may end up spanning because the MHVM is occupying the optimal NUMA node. While techniques such as virtio-balloon [17] can reclaim memory in a NUMA-aware manner, these approaches cannot ameliorate the fundamental decrease in locality when reclaiming memory from *multiple* VMs.

We quantify the potential impact of NUMA spanning by measuring the performance of representative workloads derived from Azure's internal production workloads. Table 1 summarizes the workloads and their key metrics. On a server with two NUMA nodes, we manually force the VM hosting the workload to have half of its memory in a remote NUMA node. We then run each workload 5 times. Figure 4(a) shows that performance degrades by up to 20% at the median point and almost 40% in one experiment (*SQL*). Thus, NUMA spanning can have a substantial impact.

Large pages and memory fragmentation. While large page performance is well studied [20, 35, 36], its *interaction* with memory reclamation is less well so. Today, reclamation is typically done at 4KB granularity, without the ability to enforce a specific contiguity requirement (2MB or 1GB). Therefore, reclaiming memory from a VM with fragmented guest physical address space may lead to fragmenting the host memory, which in turn prevents the host OS from allocating large or huge pages for new incoming VMs.

Figure 4(b) shows the performance loss when running workloads on a VM with fully fragmented second-level mappings (*i.e.*, all 4 KB pages). The performance of *SQL* suffers severely, but all workloads (except *Search*) see some impact showing that fragmentation is highly undesirable for cloud workloads.

4.3 Techniques to tackle harvesting challenges

We propose 4 techniques for addressing the above challenges: expediting evictions, reserving a memory buffer, balancing across NUMA nodes, and reclaiming large pages.

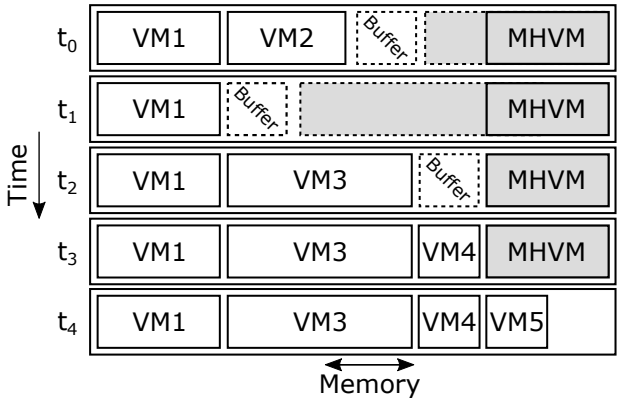


Figure 5: A MHVM dynamically changing size over time.

Expediting evictions. Harvest VMs get a warning (*e.g.*, 30 seconds in advance), if their minimum size is needed for new regular VMs. To keep creation time low, we may have to evict a MHVM more aggressively if the MHVM (1) does not return enough memory, (2) does not return memory fast enough, (3) does not return contiguous and aligned (2 MB) memory, or (4) does not return memory from the right NUMA node. In these cases, we immediately evict the MHVM to make room for a regular VM. If the VM being created is also low-priority (*e.g.*, a spot VM), we wait longer for a pending reclamation, before evicting the MHVM.

Reserving a memory buffer. To minimize regular VM creation time, we reserve a buffer of unallocated memory that cannot be harvested. Memory for a new VM can come from the buffer. However, if the buffer is not large enough for the VM, we still have to reclaim the difference from the MHVMs. For example, if we have a buffer of 16 GB and need to create a 24 GB VM, we will seek to reclaim 8 GB of memory. The buffer is replenished by reclaiming additional memory from the MHVMs off the critical creation path.

Balancing across NUMA nodes. To minimize NUMA spanning we (1) leverage the memory buffer, and (2) create MHVMs with a virtual NUMA topology and selectively assign and reclaim memory from specific NUMA nodes. The buffer reserves unallocated memory across all NUMA nodes evenly. This maximizes the likelihood that regular VMs will be fully backed by single NUMA nodes.

Reclaiming large pages. If MHVMs are allowed to return small (4 KB) pages when shrinking, this may lead to fragmentation of the host physical memory, which in turn may lead to regular VMs being assigned small pages. For this reason, we require from all MHVMs to return contiguous large pages. If the guest physical memory is fragmented, the memory manager will trade pages to honor the large page contiguity requirement, which directly impacts the reclamation speed. If for some reason the guest is not able to reclaim memory, we evict it and release its full memory. To avoid slowdowns and evictions, at MHVM boot time we create a special memory partition inside it to isolate the memory used by the harvesting framework from the rest of the guest (*e.g.*, daemons, OS). This isolation ensures that the MHVM can more easily free up contiguous large ranges (without expensive trading) to be reclaimed by the host. We implement this for Windows but our work applies to other guests that support the notion of memory partitions.

Example. Figure 5 shows a server with 256 GB of memory. It hosts 2 regular VMs with 64 GB each. At t_0, a MHVM with a minimum of

64 GB lands on the server. There are 64 GB of unallocated memory and as we leave a buffer of 32 GB, the MHVM grows to 96 GB. At t_1, VM2 finishes and the MHVM grows to 160 GB (still leaving the buffer of 32 GB). At t_2, VM3 with 96 GB lands on the server and the MHVM shrinks to its minimum (*i.e.*, 64 GB). To make room for the 96 GB, we use the 32 GB of the buffer and have to wait for the MHVM to return the remaining 64 GB. Without the buffer, VM3 would have to wait for the MHVM to return the full 96 GB. At t_3, VM4 with 32 GB lands on the server, consuming the full buffer. At t_4, VM5 with 32 GB lands in the server and the MHVM needs to be evicted to make room.

4.4 Optimizing memory resizing

Besides the above techniques, we optimize the memory resizing process by modifying the resizing driver in the guest OS, modifying the memory allocator in the hypervisor, and introducing a communication path between the host and MHVMs. We fully implement our optimizations for Windows guests but the same approach can be implemented for other guests (*e.g.*, Linux) as well.

Pre-reclamation. When the hypervisor asks the guest OS for a chunk of memory, our guest driver starts reclaiming additional chunks (*e.g.*, 128 MB) in the background. If there are no requests for a period of time (*e.g.*, 1 second), we return this pre-reclaimed memory back to the guest's memory manager. This approach gives the guest extra time to obtain large free pages, directly improving reclamation bandwidth.

Batch size. Reclaiming memory is a synchronous process between the host and the guest, where the host asks for one small chunk of memory at a time. To reduce the time to reclaim larger amounts of memory, we allow requesting larger, multi-GB chunks.

Application notification. In the default setting, the harvesting manager monitors how much memory is available at each point in time and adjusts applications accordingly. This *reactive* approach may not be efficient. To improve this, we introduce a paravirtual interface [29] where the host can notify the MHVM of the target memory size prior to requesting actual memory. This way, applications can adjust to the final target size, instead of adjusting incrementally after each batch. We find that the *proactive* approach replenishes the large page cache in the guest OS faster, making these pages readily available to the hot-remove driver.

Multiple MHVMs per server. We allow creating multiple MHVMs per server. The distribution of the harvested resources is proportional to the minimum sizes. For example, a MHVM with a minimum size of 32GB will harvest double the memory of a 16GB one (capped by the maximum specified by the user). Note that multiple MHVMs in the same server improves the overall reclamation throughput, as we can reclaim their memory in parallel.

4.5 Other considerations

Harvesting allocated but unutilized memory. The ability to harvest memory that is allocated to regular VMs but not currently utilized would increase the pool of harvestable memory. However, returning memory back to a regular VM may not be fast enough to avoid violating the strict service-level agreement for regular VMs. Also, resizing regular VMs is not acceptable, as the guest will see less memory, while transparently backing on demand can lead to serious performance consequences. A potential solution would be for VMs to explicitly give up memory that they know they will not need for a while, and request it back in advance of their use. We leave this option as future work.

Pricing. Though a detailed discussion of pricing schemes is beyond our scope, we expect MHVMs to be treated the same as Harvest VMs [4]. Specifically, we charge for the minimum size as a standard evictable VM of equal size. Users pay a discounted price (*i.e.*, by an α factor) on the additional harvested resources beyond the minimum. We tie together cores and memory at the minimum size ratio and charge for the minimum of the two. With this pricing, harvesting both cores and memory is always beneficial.

Privacy and confidentiality. On individual servers, MHVMs reveal the VM arrival and departure events. However, they do not reveal the platform's resource utilization, unless the attackers are allowed to deploy MHVMs to most servers. To avoid this, the platform can enforce a quota for MHVMs in each region. The privacy of the workloads is also protected, as MHVMs do not have any visibility into co-located VMs (memory zeroed out by the hypervisor).

Other techniques. Second-level paging by the hypervisor to swap space or a remote server could be done to reclaim memory. [14] does this when cooperative memory trading cannot happen. However, cooperative memory trading on the same node involves less performance jitter (no I/O interrupts and no second-level page faults) and is much more efficient than paging to swap space or a remote node. With second-level paging, the application will experience sudden performance degradation, while in memory harvesting as the VM is being resized, the framework adjusts accordingly, leading to less jitter for the application. Importantly, transparent second-level paging requires the ability to properly handle direct memory access by I/O devices. This technology [37] is not fully supported by the majority of I/O devices and lacks full system software support.

5 MEMORY-HARVESTING APPLICATIONS

Target applications. When operating on harvested resources, performance can be unpredictable because the amount of resources available for harvesting depends on load conditions external to the VM; a harvesting VM might even be evicted altogether. For these reasons, harvesting is a good match for batch-style workloads or applications that have loose performance requirements. Workloads that run in frameworks like Hadoop (*e.g.*, ML training [47], big-data analytics) and FaaS (*e.g.*, background event processing, data ingestion) often have those characteristics. In terms of memory harvesting specifically, workloads that can use as much memory as available for caching (*e.g.*, databases [33], distributed caches [27]) should significantly benefit. However, even these workloads must be able to tolerate the potential variability in cache sizes [41]. Prior work has explored resource and availability SLOs that reduce the unpredictability associated with resource harvesting [4, 7].

Conversely, workloads with strict performance requirements are not good matches for resource-harvesting VMs. Certain applications might also be difficult to modify to take advantage of harvested resources. For example, workloads with limited parallelism or that optimize for memory amount and placement might not be good targets.

Having to re-engineer legacy applications for harvesting is a main reason why we focus on modifying SaaS, PaaS, and FaaS frameworks that can hide those complexities. To demonstrate this, we extend a PaaS (Hadoop) and a FaaS (Azure Functions) framework to adapt them to MHVMs.

Memory-harvesting Hadoop. MH-Hadoop is our extension of Harvest Hadoop [4], which is being successfully used in production. In addition to the core changes, the harvesting manager notifies cluster-wide Resource Manager (RM) of the changes in memory size. The RM then adjusts the number of containers on the server accordingly. When less memory is available to the MHVM, the RM stops sending it new containers, and drains the running containers (waits until they complete) in the MHVM until it is within the memory budget. To avoid expedited evictions, if the draining is taking too long, MH-Hadoop may kill some of the most recently started containers. The RM also stops sending containers to soon-to-be-evicted MHVMs and treats an unexpected eviction the same as a server failure. Any running containers that get killed are later restarted by the RM. This mechanism is already part of Hadoop [52, 53]. One could also use other mechanisms like interruptible tasks [13].

Note that Hadoop hosts a variety of batch-style workloads that are often run on Spot VMs. Hadoop also often underlies Spark [54] and MH-Hadoop can run more or larger Spark executors depending on the harvested memory. For example, the *ML* workload in Table 1 which uses TensorFlow [1] runs on Hadoop.

Memory-harvesting FaaS. Serverless frameworks have already been adapted to run on Harvest VMs [58]. However, harvesting memory requires additional extensions. We build memory-harvesting FaaS (MH-FaaS) as an extension to Azure Functions [30]. The offering is similar architecturally to OpenWhisk [34], and consists of a *controller* that forwards function invocations to the MHVMs. Each MHVM runs an *invoker* that executes the function invocations in containers.

In our MH-FaaS, each harvesting manager notifies the controller of the changes in cores and memory, so the controller can distribute function invocations across the MHVMs accordingly. The controller stops sending invocations to MHVMs that are shrinking, while the invoker drains currently running containers and (if necessary) destroys idle containers that are simply waiting for other invocations [15, 42]. The controller also stops sending new invocations to soon-to-be-evicted MHVMs and treats unexpected evictions as failures. FaaS runs short-duration functions in relatively small containers [42]. Thanks to memory harvesting, MH-FaaS can maintain a larger function instance cache and reduce cold starts [58].

6 EVALUATION

6.1 Methodology

Workloads. In our experiments, the regular VMs host the common cloud applications presented in Table 1. These applications run in both Windows (Windows Server 2016 and 2019) and Linux (Ubuntu 18.04 and 20.04). To evaluate the performance degradation caused by harvesting, we monitor their key performance metrics in three scenarios: (1) with an idle MHVM which represents the best case, (2) with a MHVM running a CPU bully application which represents the worst case, and (3) with MH-Hadoop running in the MHVM which represents the common case. The bully application saturates every virtual CPU while using very little memory. MH-Hadoop runs MapReduce jobs with many memory-hungry map workers. These applications run on Windows Server 2019. We also run MH-FaaS on Ubuntu 20.04.

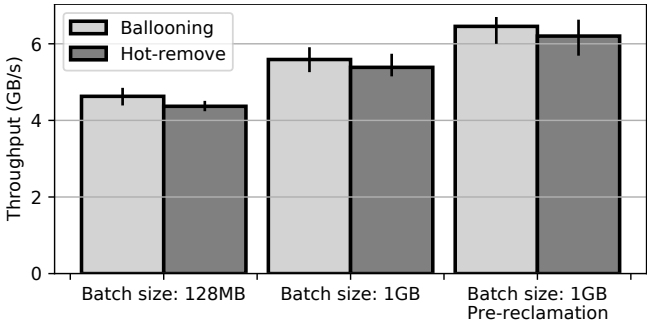


Figure 6: Ballooning in Windows Server 2019 is slightly better than the hot-remove; both can be improved with driver pre-reclamation and large batch sizes.

Servers. Our experiments run on production servers with 2 NUMA nodes, each with 20 cores for a total of 80 logical (hyper-threading) processors, and 512 GB of memory. We reserve 64 GB to run the server agents (including our MHVM code) and reserve 224 GB per NUMA node just for VMs. For the hypervisor, unless stated otherwise, we use Hyper-V from Windows Server 2019 [28].

6.2 Memory harvesting mechanism

We start with synthetic experiments to evaluate all the changes that MHVMs could potentially see in production. Once we instantiate a MHVM, our benchmarking setup programmatically adjust its memory allocation, measuring the time to complete the resize, or if it was unsuccessful. This level of control allows us to capture key metrics such as memory fragmentation and NUMA locality.

Sensitivity to the MHVM size. We take idle MHVMs and shrink them using hot-remove from 128, 96, 64, 32 and 16 GB of RAM to 96, 64, 32, 16, 8 and 4 GB. The reclamation throughput of just above 4 GB/s is consistent across resizes. Resizing to smaller memory sizes (*e.g.*, 4 GB target) is marginally slower as there is so little memory in the VM that it takes longer for the guest to reclaim each batch.

Interestingly, reclamation throughput scales linearly in the number of VMs. When we are reclaiming memory from 4 MHVMs in parallel, the reclamation throughput increases 4×. Such increase in throughput is because the reclamation from each individual VM is performed by a separate host worker. On the other hand, adding memory through hot-add is much faster and is higher than 40 GB/s. From now on, we focus on the throughput of shrinking a MHVM, because it is on the critical path when launching new VMs.

Sensitivity to the host OS and guest driver changes. Running reclamation using the same server with the Hyper-V [28] or KVM [22] hypervisors shows comparable results. We compare Windows Server 2016 and 2019 with Ubuntu 18.04 and 20.04 running in the MHVM and the reclamation throughput is comparable. The main differences come from the configuration of the reclamation driver running in the MHVM and how the host requests memory from the guest.

Figure 6 shows the memory reclamation throughput depending on the driver setting. The baseline *hot-remove* and *ballooning* show

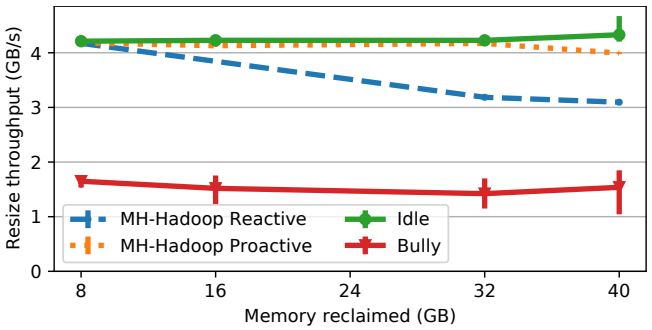


Figure 7: Memory shrinking throughput depending on the workload running in the VM.

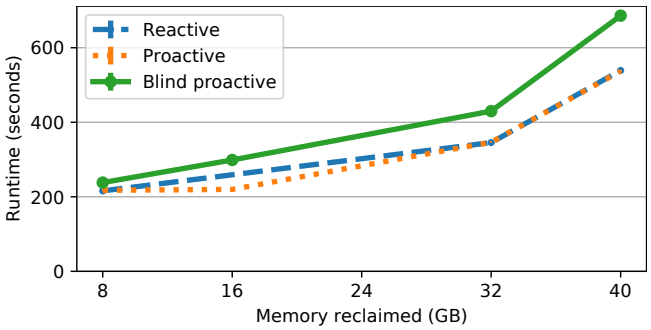


Figure 8: Runtime of MH-Hadoop jobs when reclaiming memory from a MHVM. The runtime increases as we reclaim more memory. "Reactive" and "proactive" show similar performance.

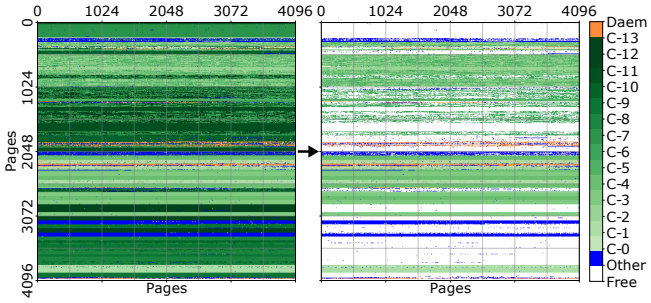
an average throughput of 4.4 GB/s and 4.6 GB/s, respectively. *Hot-remove* is slightly slower as it requires updating the guest physical address space on top of *ballooning*. However, from our experience most users prefer *hot-remove* as they can track the memory size. The figure also shows the impact of our optimizations using large batch sizes and *pre-reclamation*. *Hot-remove* achieves 5.6 GB/s with a batch size of 1 GB and 6.2 GB/s using both 1 GB batch size and pre-reclamation of up to 2 GB.

Sensitivity to the workload. Figure 7 shows the throughput when shrinking a MHVM running different workloads. As expected, the idle VM achieves the best throughput with 4.25 GB/s, while the CPU bully with 1.5 GB/s is the worst case.

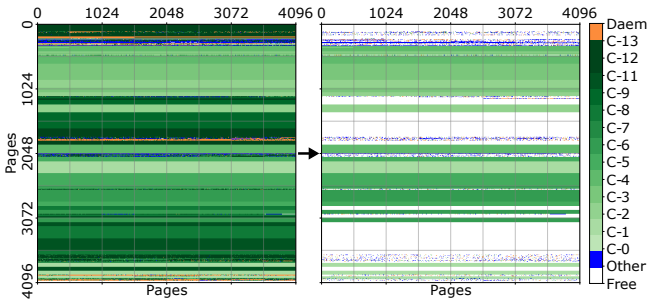
To evaluate the benefit of the *application notification* mechanism, we compare the performance of MH-Hadoop using notifications (*i.e.*, proactive) and adjusting memory incrementally based on the amount of free memory (*i.e.*, reactive). The *proactive* resizing of MH-Hadoop matches the throughput of the idle VM. This is because MH-Hadoop releases all the reclaimed memory at once, ensuring most of the memory is readily available ahead of each resize batch. The *reactive* model does not perform nearly as well as it slowly releases memory over time, leaving the driver with more work to do when servicing resize requests.

Figure 8 shows the time to complete the MapReduce jobs in a MHVM as a functions of the reclaimed memory size with multiple strategies. *Blind proactive* does not monitor the actual memory utilization and assigns a fixed percentage of the VM to MH-Hadoop leading to the longest runtimes. The *reactive* model achieves higher job throughput by pushing the memory limits of the system. Our *proactive* approach combines these two and gets the benefits of both, trying to maximize guest memory usage but shrinking on demand upon a host notification. This gives us the best reclamation throughput with the lowest performance impact on the workload.

Memory fragmentation. Memory fragmentation inside the MHVM does not necessarily impact the reclamation bandwidth, but it can have severe consequences for regular VMs if reclamation is not done at large page granularity. Reclaiming large pages is easier if the harvesting application releases memory in large chunks. To confirm this, we run two identical workloads, forcing MH-Hadoop to allocate large or small pages. Figure 9 shows the process owning each of the 16M 4-KB pages in a 64-GB MHVM running the MH-Hadoop workload. This memory snapshot includes the MH-Hadoop daemons, the containers running tasks, all other processes, and the free pages that are ready to be reclaimed. The



(a) Using 4-KB page allocations shrinking to 32 GB.



(b) Using 2-MB large page allocations shrinking to 32 GB.

Figure 9: View of the memory of a 64-GB MHVM running MH-Hadoop with small and large pages. Each point is a 4-KB page and its color indicates the owning process. Each square represents 1 GB of memory ($4KB \times 512 \times 512$).

left side shows the starting point with MH-Hadoop running 14 tasks in 14 containers and taking the full 64 GB. This leaves only 46 MB and 36 MB free with (Figure 9(a)) and without large pages (Figure 9(b)), respectively. We then reclaim 32 GB of memory from the MHVM (right side), making MH-Hadoop reduce its memory footprint and preempt 7 tasks. Without large pages, there are 35.9 GB of free memory of which 17.8 GB are contiguous 2-MB chunks. Using large pages, there are 39.8 GB of free memory of which 22.9 GB are contiguous. Interestingly, 19 out of these 22.9 GB are 1-GB aligned chunks while there are none using 4-KB pages. If we were to reclaim 1 GB (*i.e.*, huge pages), MH-Hadoop using large page allocations would enable memory to be reclaimed much faster.

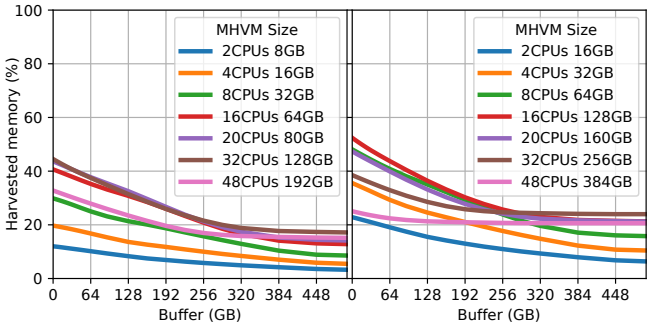


Figure 10: Harvested memory varying the MHVM minimum size and the buffer. MHVMs of 8 CPUs and 32 GB of memory harvest around 20% of the unallocated memory with a buffer of 192 GB.

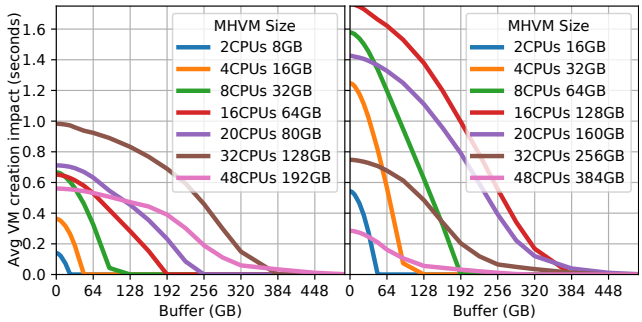


Figure 11: VM creation time impact varying the minimum size and the buffer. A buffer of 192 GB has no impact when harvesting with a minimum of 64 GB or less.

6.3 Cloud platform analysis

To evaluate harvesting at fleet scale, we use the Azure traces of the servers characterized in Section 3. To model the reclamation throughput and other characteristics, we use the results from the previous set of experiments. To understand the effect of placing MHVMs in the available space, we evaluate: (1) additional time to create regular VMs, (2) amount of harvested resources, and (3) forced NUMA spanning.

In this analysis, we assume that once a MHVM is placed, it can grow up to 4× its minimum size. For example, a MHVM created with a minimum of 4 cores and 32 GB of memory can grow all the way to 16 cores and 128 GB.

Harvested memory. Figure 10 shows how much of the unallocated platform memory we can harvest depending on the minimum size of the MHVM and the buffer. MHVMs with 4 CPUs and 16 GB harvest around 10% of the platform memory with a buffer of 256 GB. This amount of unallocated memory harvesting at fleet scale means a significant efficiency improvement and an additional source of revenue for the cloud platform. Without any buffer, we harvest the most memory and the larger the buffer is, the less we can harvest. Smaller MHVMs can harvest less memory but as the buffer grows, the gap between smaller and larger MHVMs shrinks. MHVMs with a higher memory to CPU ratio also harvest more memory.

VM creation time. Figure 11 shows the average time we add to the creation of regular VMs depending on the minimum size of the MHVM and the buffer. The larger the buffer is, the less impact to the VM creation time. At a buffer of 192 GB, we can harvest without any impact with a MHVM of minimum size 64 GB. In addition, the smaller the minimum MHVM size, the less impact on regular VM creation times. We see substantially less impact with a ratio of 4 GB of memory per CPU (left of Figure 11) compared to 8 GB per CPU (right of Figure 11).

NUMA spanning. Figure 12 shows the percentage of VMs that suffer NUMA spanning when we place a MHVM of 8 cores and 32 GB of memory, as a function of buffer size. Using a MHVM that is not NUMA-aware (the blue curve in the figure) causes more spanning in regular VMs. The number of affected VMs decreases as we increase the buffer size, but will never match the baseline without harvesting (dotted line). If the MHVM uses virtual NUMA, the forced spanning is reduced; by leaving a buffer of 128 GB, harvesting matches the baseline and ensures no additional VMs will see NUMA spanning.

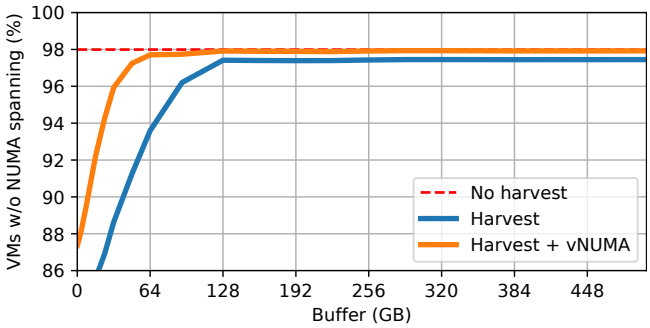


Figure 12: Regular VMs without NUMA spanning when placing using MHVMs of 8 cores and 32 GB of memory with and without virtual NUMA, as a function of buffer size.

Takeaways. A 192 GB buffer and MHVMs with a minimum of 64 GB provide a good balance between the amount of harvested memory and the impact to regular VMs. Larger buffers and smaller MHVMs are safer but do not harvest as much memory. Cloud platforms need to consider these tradeoffs when implementing MHVMs.

6.4 Test in production

To evaluate the effectiveness of MHVMs, we use VM arrivals and departures from production servers located in one of the most popular clusters on July 20, 2021 from noon to 1 PM. We randomly select 10 servers based on their utilization, excluding uninteresting cases: empty servers (*i.e.*, MHVMs would always run with the maximum memory), full servers (*i.e.*, MHVMs cannot run), and servers without arrivals or departures (*i.e.*, MHVMs would not change size). We reserve identical, empty servers in the same cluster and replay the VM arrivals and departures for each of the selected traces.

Each of these VMs runs one of the cloud workloads from Table 1 based on their VM size. We then fill the resource holes left by the regular VMs using MHVMs. The servers are configured with a buffer of 192 GB. The MHVMs have a minimum size of 4 virtual CPUs and 32 GB of memory and can grow up to twice their minimum size. We run at most one MHVM at a time on each server. If a MHVM gets evicted, we place a new MHVM once the resources become available again. Overall, the 10 servers host 144 regular VMs and we can place 34 MHVMs.

Harvested resources. Table 2 shows the average size of the MHVM for each of the 10 servers. The 34 MHVMs shrank a total of 15

Server	MHVMs	Memory (GB)	Running (seconds)	Evict	Shrink	Grow
1	1	54.2	3600	0	3	2
2	3	47.9	2990	2	1	1
3	2	18.6	1677	1	2	1
4	4	47.5	2813	3	1	1
5	5	19.4	1682	5	6	5
6	8	17.0	1283	7	1	1
7	3	38.3	2203	2	1	0
8	2	47.3	2659	2	0	0
9	3	46.3	2607	2	0	0
10	3	32.9	1850	3	0	0
Total	34	38.9	2409	27	15	11

Table 2: Servers over 1 hour, running MHVMs when possible.

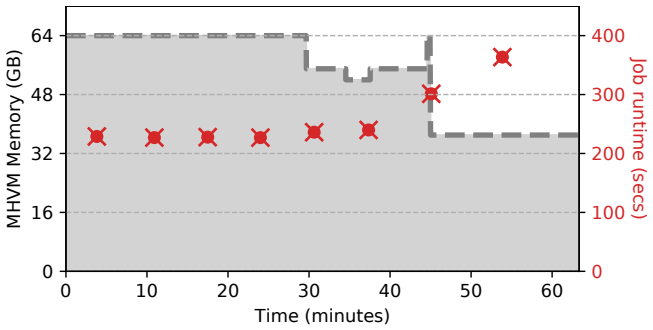


Figure 13: Performance of MH-Hadoop jobs as the MHVM grows and shrinks due to co-located regular VMs arriving and departing.

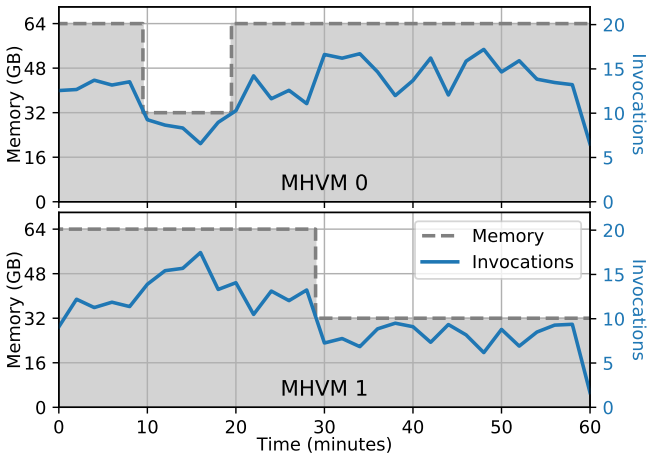


Figure 14: Performance of MH-FaaS running in 2 MHVMs.

times, achieving a resize throughput of 8.5 GB/s when running MH-Hadoop jobs, and 10.6 GB/s when idle. The MHVMs also grew a total of 11 times, at a much higher throughput of 49.3 GB/s. Server 1 has a MHVM running for the full hour with 3 shrinks and 2 grows, allowing it to harvest an average of 54.2 GB of memory. Server 6 shows the highest number of evictions (7) over 1283 seconds for an average of 17 GB harvested during its lifetime. On average, our MHVMs run for 2409 seconds and harvest 38.9 GB during that time. This experiment represents a wide variety of scenarios for MHVMs.

VM creation time. Given that the buffer of 192 GB in size allows reclaiming memory from the MHVMs off the critical path, we observe no additional time to create any of the regular VMs. To validate the effectiveness of the buffer, we also run the traces without any buffer and find that regular VMs do see additional creation time. For example, in Server 5, MHVMs add a total of 5.9 seconds to the creation time of 6 regular VMs (out of 15). The lack of the buffer doubles the amount of memory harvested by the MHVMs running on this server. This is an instance of the tradeoff between the amount of harvested resources and the impact to VM creation time (Figures 10 and 11). Cloud providers will likely prefer not having any impact on creation time at the cost of harvesting less.

Workload performance. In our production tests, the performance of cloud workloads running in regular VMs does not degrade. This is expected as we do not observe any additional NUMA spanning or memory fragmentation. To validate that our implementation avoids additional NUMA spanning, we disable surfacing the NUMA topology to the MHVMs (vNUMA). With this setup, we are able to force NUMA spanning for 2 regular VMs in Servers 3 and 5. This is consistent with the analysis in Figure 12 and shows that a properly-sized buffer alone cannot eliminate NUMA spanning.

We do not observe any memory fragmentation as the MHVM is not allowed to return non-contiguous 2 MB memory. We also do not observe any MHVM evictions caused by the guests not returning contiguous 2 MB chunks on time. This is largely due to the harvesting workloads allocating and releasing large pages, and the application notifications enabling the guest to release pages on time (before the reclamation starts).

MHVM evictions. By default, MHVMs are evicted when their minimum resources are needed for running regular VMs. Our experiments show 27 of these evictions and each such eviction is due to all the cores being allocated. In such cases, the MHVM must be evicted to release the cores so that the incoming regular VM can run. MHVMs can also be evicted if they do not release enough memory quickly enough. To evaluate this, we run a non-cooperative workload in the MHVM that does not leave enough free memory to reclaim. Our implementation identifies that the MHVM is not returning enough memory and evicts it within 30 seconds.

MH-Hadoop. Inside each MHVM placed on servers in Table 2, we run MH-Hadoop. Figure 13 shows the run for Server 1, where the memory size of the MHVM varies over time due to the arrivals and departures of regular VMs. The performance of MH-Hadoop is affected by the memory allocated to the MHVM. Initially, the MHVM reaches the maximum of 64 GB and then shrinks and grows during the second half hour, ending at 37 GB. During the first half hour, the MH-Hadoop jobs take around 230 seconds. As the MHVM shrinks in the second half, MH-Hadoop needs to cut the number of containers in half. This translates into a job runtime increase to 350 seconds (*i.e.*, 52.2% degradation). There are no job failures as MH-Hadoop is successful at adjusting to the resource variability.

MH-FaaS. We run a subset of the FaaS workload described by Shahrad *et al.* [42] on MH-FaaS using Spot VMs, Harvest VMs (*i.e.*, harvesting cores but not memory), and MHVMs with a minimum of 4 CPUs and 32 GB of memory. With the same number of VMs, harvesting allows us to host more containers and execute more function invocations. Harvest VMs can support 2.5× more invocations than Spot VMs, while MHVMs can handle 3× more invocations than Spot and 20% more than Harvest VMs.

To evaluate the ability of MH-FaaS to adjust to changes in memory size, we run the workload with 2 MHVMs running on

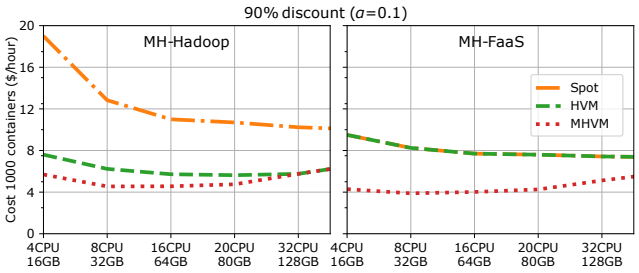


Figure 15: Costs of running MH-Hadoop and MH-FaaS in MHVMs at 90% discount for harvested resources, compared to Spot and Harvest VMs. The cost of using regular VMs (not shown) is at least 5× higher than that for Spot VMs.

production servers and compare it against *vanilla FaaS*, which is harvesting-oblivious. Figure 14 shows the memory assigned along with the function invocations running in each MHVM. MH-FaaS balances the load according to their available resources to avoid performance degradation when one shrinks. In minute 10, MH-FaaS shifts the load to MHVM 1 without failing any invocation. In contrast, vanilla FaaS (not shown) does not adjust to changes in memory size and the OS kills containers when memory shrinks, leading to failed invocations around minutes 10 and 30.

Takeaways. The 192-GB buffer is successful at not delaying regular VM creation. The buffer along with exposing vNUMA to the guest OS avoid NUMA spanning in the regular VMs, and our driver avoids any memory fragmentation in the regular VMs.

6.5 Cost comparisons

To compute the savings users can accrue from MHVMs, we compare the costs of running the same workload using regular VMs, Spot VMs, Harvest VMs, and MHVMs. For this comparison, we compute the average per-server number of cores and amount of memory available from the Azure characterization in Section 3. To prevent impact to regular VM creation time and NUMA spanning, we set the memory buffer to 192 GB. We calculate the cheapest VM size for each VM type and compare total costs. We assume the public costs of $0.096 per (core×hour) for a VM with 4 GB of memory per core and $0.126 per (core×hour) for 8 GB per core. The discount for Spot VMs is 80%.

Figure 15 shows the costs per hour to host MH-Hadoop and MH-FaaS with 1000 containers. The X-axis shows the size of the Spot VMs and the minimum size for the Harvest VMs and MHVMs. We assume that the latter VMs can grow to a maximum of 4× their minimum size. The discount for the harvested resources beyond the minimum size is 90% (*i.e.*, $\alpha = 0.1$). Regular VMs are much more expensive and are well outside the figure's cost range.

MH-Hadoop. To run tasks, we use containers of 1 CPU and 2 GB of memory. In addition, we reserve 2 CPUs and 4 GB for the node daemons and configure the Harvesting Manager to leave a buffer of 2 CPUs and 2 GB of memory. With this configuration, the savings from regular VMs to Spot VMs is 79.9% (*i.e.*, the Spot discount with the eviction overheads). Figure 15 shows that at a 90% discount for harvested resources, Harvest VMs are 88.8% cheaper and 90.9% for MHVMs; this translates into 19.1% savings from Harvest VMs to MHVMs. For a 100% discount ($\alpha = 0$), the MHVM savings are 92.8% compared to regular VMs and 21.4% compared to Harvest VMs.

MH-FaaS. To execute functions, we use containers with 0.5 CPU and 3 GB of memory and reserve 2 CPUs and 4 GB for the daemons running the FaaS platform. The Harvesting Manager leaves a buffer of 2 CPUs and 2 GB of memory. Figure 15 shows that at a 90% discount, MHVMs are 90.3% and 35.6% cheaper than regular and Harvest VMs, respectively. With a discount of 100%, MHVMs are 92.0% and 45.3% cheaper than regular and Harvest VMs.

In these results, Harvest VMs and Spot VMs incur the same costs because the containers' size makes the workload memory-bound, *i.e.* adding more cores does not enable running more containers in each VM. Zhang *et al.* [58] found cost savings from Harvest VMs when containers consume less memory per core.

Other workloads. With MH-Hadoop and MH-FaaS, we observe that the advantages of MHVMs are sensitive to the characteristics of the workloads. But there are workloads with other memory to CPU ratio requirements as well. The more memory-hungry the workload is, the cheaper MHVMs are compared to Harvest VMs.

Sensitivity to the discount α. As we have seen, increasing the discount for the harvested resources makes MHVMs an even better option compared to Harvest VMs. When the platform offers no discount, the costs for Spot VMs, Harvest VMs, and MHVMs are almost identical (Spot VMs still involve more evictions).

Performance benefits. Figure 15 fixes the amount of resources available (1000 containers) and studies the cost of multiple workload and VM size combinations. However, the reverse is also possible: fix the budget and calculate the number of containers (*i.e.*, resources) that each combination gets. This is a direct proxy for performance (Figures 13 and 14). These results are directly proportional to the ones in Figure 15: MH-Hadoop and MH-FaaS get better performance when running on MHVMs with the same budget.

Takeaways. MHVMs provide savings for memory-hungry workloads, especially with highly discounted harvested resources. Given our results, we expect users to migrate from Spot VMs and Harvest VMs to MHVMs for our frameworks, preferring the cheaper resources, more memory, and lower eviction rates.

7 CONCLUSIONS

In this paper, we proposed harvesting unallocated memory from cloud platforms. Our characterization of Azure showed 30%-50% of server memory available to be harvested. Thus, we proposed Memory-Harvesting VMs (MHVMs) and addressed the key challenges in runtime VM memory resizing, namely additional VM creation time, NUMA locality, and large page fragmentation. To demonstrate the use of MHVMs, we also extended Hadoop and a FaaS platform to adapt to changes in memory size. Our results showed that MHVMs can successfully harvest more than 20% of the unallocated memory in Azure with no impact on regular VMs. Moreover, we showed that our extensions of Hadoop and FaaS allow these workloads to run transparently on MHVMs at up to 93% lower cost than using regular VMs. We conclude that memory harvesting and MHVMs show great potential for practical deployment.

ACKNOWLEDGEMENTS

We thank Lieven Eeckhout (our shepherd), the anonymous reviewers, Ethan Young, and Pantea Zardoshti for their many helpful comments and suggestions.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A System for Large-Scale Machine Learning. In *OSDI*.

[2] Mohammad Agbarya, Idan Yaniv, and Dan Tsafrir. 2018. Memomania: From Huge to Huge-Huge Pages. In *SYSTOR*.

[3] Amazon Elastic Compute Cloud. 2019. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.

[4] Pradeep Ambati, Íñigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *OSDI*.

[5] Microsoft Azure. 2019. Introducing B-Series, Our New Burstable VM Size. https://azure.microsoft.com/en-us/blog/introducing-b-series-our-new-burstable-vm-size/.

[6] Microsoft Azure. 2020. Azure Spot Virtual Machines. https://azure.microsoft.com/en-us/pricing/spot.

[7] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. 2014. Long-term SLOs for reclaimed cloud computing resources. In *SoCC*.

[8] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. 2013. Working set-based physical memory ballooning. In *ICAC*.

[9] Amazon Elastic Compute Cloud. 2019. Burstable Performance Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html.

[10] Google Cloud. 2020. Preemptible VM Instances. https://cloud.google.com/compute/docs/instances/preemptible.

[11] Standard Performance Evaluation Corporation. 2015. SPECjbb 2015. https://www.spec.org/jbb2015/.

[12] Eli Cortez, Anand Bonde, Alexandre Muzi, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP*.

[13] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP*.

[14] Alexander Fuerst, Ahmed Ali-Eldin, Prashant Shenoy, and Prateek Sharma. 2020. Cloud-scale VM-deflation for Running Interactive Applications On Transient Servers. In *HPDC*.

[15] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *ASPLOS*.

[16] Ligang He, Deqing Zou, Zhang Zhang, Chao Chen, Hai Jin, and Stephen A Jarvis. 2014. Developing resource consolidation frameworks for moldable virtual machines in clouds. *Future Generation Computer Systems* 32 (2014), 69–81.

[17] David Hildenbrand and Martin Schulz. 2021. virtio-mem: Paravirtualized Memory Hot(Un)Plug. In *VEE*.

[18] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2018. HUB: Hugepage ballooning in kernel-based Virtual Machines. In *MEMSYS*.

[19] Jinchun Kim, Viacheslav Fedorov, Paul V Gratz, and AL Narasimha Reddy. 2015. Dynamic memory pressure aware ballooning. In *MEMSYS*.

[20] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with ingens. In *OSDI*.

[21] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *EuroSys*.

[22] Linux. 2021. KVM. https://www.linux-kvm.org/.

[23] Haikun Liu, Hai Jin, Xiaofei Liao, Wei Deng, Bingsheng He, and Cheng-zhong Xu. 2014. Hotplug or ballooning: A comparative study on dynamic memory management techniques for Virtual Machines. *IEEE Transactions on parallel and distributed systems* 26, 5 (2014), 1350–1363.

[24] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. In *ISCA*.

[25] Zoltán Ádám Mann. 2016. Multicore-aware Virtual Machine Placement in Cloud Data Centers. *Transactions on Computers* 65, 11 (2016), 3357–3369.

[26] Ming Mao and Marty Humphrey. 2012. A Performance Study on the VM Startup Time in the Cloud. In *CLOUD*.

[27] Memcached. 2021. Memcached. https://www.memcached.org/.

[28] Microsoft. 2016. Hyper-V Technology Overview. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview.

[29] Microsoft. 2019. Hyper-V Integration Services. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/integration-services.

[30] Microsoft. 2021. Azure Functions. https://azure.microsoft.com/en-us/services/functions.

[31] Microsoft. 2021. Bing. https://www.bing.com/.

[32] Microsoft. 2021. Power BI. https://powerbi.microsoft.com/en-us/.

[33] Microsoft. 2021. SQL Server. https://www.microsoft.com/en-us/sql-server.

[34] OpenWhisk. 2021. Apache OpenWhisk Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.

[35] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *ASPLOS*.

[36] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *ASPLOS*.

[37] PCI-SIG. [n. d.]. Address Translation Services, Revision 1.1. https://composter.com.ua/documents/ats_r1.1_26Jan09.pdf.

[38] Shlomit S Pinter, Yariv Aridor, Steven S Shultz, and Sergey Guenender. 2008. Improving machine virtualisation with 'hotplug memory'. *International Journal of High Performance Computing and Networking* 5, 4 (2008), 241–250.

[39] The Next Platform. 2021. CXL and Gen-Z Iron Out A Coherent Interconnect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/. accessed 5 May 2021.

[40] Kaveh Razavi and Thilo Kielmann. 2013. Scalable Virtual Machine Deployment Using VM Image Caches. In *SC*.

[41] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. 2013. Application Level Ballooning for Efficient Server Consolidation. In *EuroSys*.

[42] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX ATC*.

[43] Prateek Sharma, Ahmed Ali-Edlin, and Prashant Shenoy. 2019. Resource Deflation: A New Approach For Transient Resource Reclamation. In *EuroSys*.

[44] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. 2013. Optimizing Google's warehouse scale computers: The NUMA experience. In *HPCA*.

[45] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *EuroSys*.

[46] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *EuroSys*.

[47] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. 2020. Spotnik: Designing Distributed Machine Learning for Transient Cloud Resources. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[48] Carl A Waldspurger. 2002. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 181–194.

[49] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *EuroSys*.

[50] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin Yousif. 2007. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI*.

[51] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *ISCA*.

[52] Apache Hadoop YARN. [n. d.]. Dynamic Resource Configuration. https://issues.apache.org/jira/browse/YARN-999.

[53] Apache Hadoop YARN. [n. d.]. In case of long running tasks, reduce node resource should balloon out resource quickly by calling preemption API and suspending running task. https://issues.apache.org/jira/browse/YARN-999.

[54] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. In *HotCloud*.

[55] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. CompuCache: Remote Computable Caching using Spot VMs. In *CIDR*.

[56] Qi Zhang, Ling Liu, Jiangchun Ren, Gong Su, and Arun Iyengar. 2016. iBalloon: Efficient VM Memory Balancing as a Service. In *ICWS*.

[57] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *EuroSys*.

[58] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Sameh Elnikety Rodrigo Fonseca, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *SOSP*.

[59] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *OSDI*.

[60] Weiming Zhao, Zhenlin Wang, and Yingwei Luo. 2009. Dynamic Memory Balancing for Virtual Machines. *SIGOPS Operating Systems Review* 43, 3 (2009), 37–47.