Mortar: Filling the Gaps in Data Center Memory

Jinho Hwang[†] Ahsen Uppal Timothy Wood H. Howie Huang

[†]IBM T.J. Watson Research Center The George Washington University

jinho@us.ibm.com {auppal, timwood, howie}@gwu.edu

Abstract

Data center servers are typically overprovisioned, leaving spare memory and CPU capacity idle to handle unpredictable workload bursts by the virtual machines running on them. While this allows for fast hotspot mitigation, it is also wasteful. Unfortunately, making use of spare capacity without impacting active applications is particularly difficult for memory since it typically must be allocated in coarse chunks over long timescales. In this work we propose repurposing the poorly utilized memory in a data center to store a volatile data store that is managed by the hypervisor. We present two uses for our Mortar framework: as a cache for prefetching disk blocks, and as an application-level distributed cache that follows the memcached protocol. Both prototypes use the framework to ask the hypervisor to store useful, but recoverable data within its free memory pool. This allows the hypervisor to control eviction policies and prioritize access to the cache. We demonstrate the benefits of our prototypes using realistic web applications and disk benchmarks, as well as memory traces gathered from live servers in our university's IT department. By expanding and contracting the data store size based on the free memory available, Mortar improves average response time of a web application by up to 35% compared to a fixed size memcached deployment, and improves overall video streaming performance by 45% through prefetching.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management

General Terms Design; Experimentation; Performance

Keywords Memory Management; Virtualization; Memcached; Disk Prefetching

VEE '14, March 01–02, 2014, Salt Lake City, UT, USA.

http://dx.doi.org/10.1145/2576195.2576203

1. Introduction

Cloud data centers can comprise thousands of servers, each of which may host multiple virtual machines (VMs). Making efficient use of all those server resources is a major challenge, but a cloud platform that can obtain better utilization can offer lower prices for a competitive advantage. A resource such as the CPU is relatively simple to manage because it can be allocated on a very fine time scale, greatly simplifying how it can be shared among multiple VMs. Memory, however, typically must be allocated to VMs in large chunks at coarse time scales, making it far less flexible. Since memory demands can change quickly and new VMs may frequently be created or migrated, it is common to leave a buffer of unused memory for the hypervisor to manage. Even worse, operating systems have been designed to greedily consume as much memory as they can—the OS will happily release the CPU when it has no tasks to run, but it will consume every memory page it can for its file cache. The result is that many servers have memory allocated to VMs that is inefficiently utilized, *and* have regions of memory left idle so that the machine can be ready to instantiate new VMs or receive a migration.

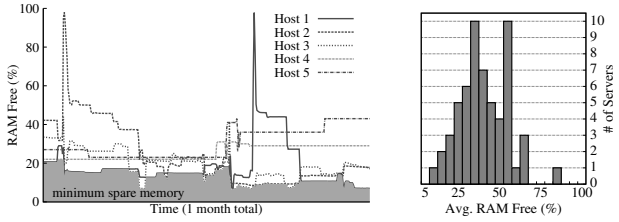


Figure 1: The amount of free memory on a set of five hosts varying over time (left), and the histogram on 58 servers from our university's data center (right).

To illustrate this inefficiency, we have gathered four months of memory traces from over fifty servers within our university's IT department. Each server is used to host an average of 15 VMs running a mix of web services, domain controllers, business process management, and data warehouse applications. The servers are managed with VMware's Distributed Resource Management software [24], which dynamically reallocates memory and migrates virtual machines based on their workload needs. Figure 1 shows the amount of memory left idle on a set of five representative machines

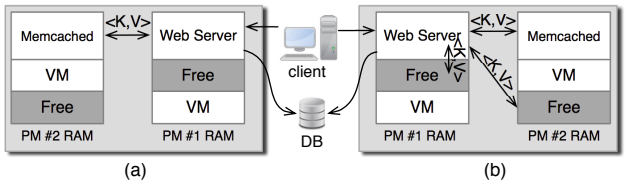


Figure 2: Physical RAM map shows how a physical machine (PM) composes its memory. (a) Traditional Memcached uses only dedicated memcached space for cache; (b) Mortar uses all the spare memory in the whole system.

over the course of a month, and the histogram of the amount of free memory on the full set of 58 servers, ignoring maintenance periods where VMs have not yet been started and nearly all memory is free. We find that at least half of the machines have 30% or more of their memory free. This level of overprovisioning was also shown in the resource observations from [4]. Clearly it would be beneficial to make use of this spare memory, but simply assigning it back to the VMs does not guarantee it will be used in a productive way. Further, reallocating memory from one VM to another can be a slow process that may require swapping to disk.

To improve this situation, we present the design of Mortar, a system that enhances the Xen hypervisor to pool together spare memory on each machine and expose it as a volatile data cache. When a server has spare memory capacity, VMs are free to add data to the hypervisor managed cache, but if memory becomes a constrained resource, the hypervisor can immediately evict objects from the cache to reclaim space needed for other VMs. This grants the hypervisor far greater control over how memory is used within the data center, and improves performance by making opportunistic use of any spare memory available.

We present two example usages for the Mortar framework. Our first prototype aggregates free memory throughout the data center for use as a distributed cache following the standard memcached protocol. This allows unmodified web applications to achieve performance gains by opportunistically using spare data center memory. Next, we demonstrate how Mortar can be used at the OS-level to transparently cache and prefetch disk blocks for applications. Prefetching is an ideal candidate for Mortar's volatile data store because the aggressiveness of the algorithm can be tuned based on the amount of free memory available.

The contributions of this paper are as follows:

- A framework for repurposing spare system memory that otherwise would be idle or poorly utilized.
- A prototype disk prefetching system that aggressively reads disk blocks into spare hypervisor memory to reduce the latency of future disk reads.
- An enhanced memcached server that can utilize this hypervisor controlled memory to build a distributed application-level cache accessible by web applications.
- Cache allocation and replacement algorithms for prioritizing access to spare memory and balancing the need to retain hot data in the cache against the goal of being able to immediately reclaim memory for other uses.

We have thoroughly evaluated Mortar using microbenchmarks, realistic web applications, and disk access traces. Our results demonstrate that Mortar incurs an overhead under 0.03ms on individual read accesses, and illustrates the benefit of making use of all free memory in a data cecnter. Our fast cache release algorithm can reclaim gigabytes of memory within 0.1ms. In experiments driven by real server memory traces, Mortar improves web performance by over 35% by using a spare memory based cache. When using only 500MB of idle server memory for a prefetch cache, Mortar makes disk reads in an OLTP benchmark three times faster.

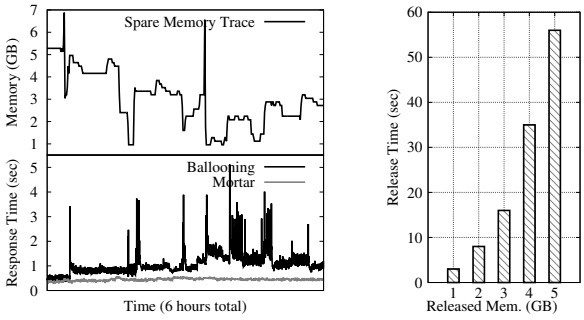
2. Background and Motivation

In this work we assume Mortar is run in a public or private cloud environment that makes use of a virtualized infrastructure to adapt quickly to different user demands. As is now common, we assume that dynamic resource provisioning techniques [3, 11, 19, 23–25] are frequently readjusting resource shares for virtual machines based on their workloads. Even in these automated systems, overprovisioning is still common since some spare capacity is left on each machine to handle rising workloads locally without resorting to more expensive VM migrations.

Ideally, this spare capacity would be opportunistically used, but then freed when it is needed for a more important purpose. For resources such as CPU time, this can be easily accomplished using existing CPU schedulers that can assign weights for different VMs and can adjust scheduling decisions on the order of milliseconds. Unfortunately, memory cannot be reassigned as efficiently or as effectively as CPU shares.

There are two challenges that prevent memory from being used as a flexible resource like CPU time. First, memory is generally only helpful if it is allocated in large chunks over coarse time scales (i.e., minutes or hours). If a VM has processing to do, it can immediately make use of more CPU time, but an increased memory allocation can take time to fill up with useful data. Further, rapidly increasing and decreasing a VM's memory share can lead to disastrous swapping. The second challenge is that adjusting a VM's memory share generally has an unpredictable impact on performance. This is partly because operating systems have been designed to greedily hoard whatever memory they can make use of. Over time, a VM will consume any additional memory pages it is given for its disk cache, but this will not necessarily have a significant impact on application-level performance.

One approach that has gained popularity for directly translating more memory into better performance is the use of in-memory application-level caches such as memcached. Many web applications, such as Wikipedia, Flickr, and Twitter, use memcached to store volatile data such as the results



(a) Web Response Time (b) Swap Time

Figure 3: Ballooning vs. Mortar

of database queries, allowing for much faster client response times as depicted in Figure 2(a). Each memcached node holds a simple key-value based data store, and these nodes are then grouped together to create a distributed in-memory cache. However, memcached works by allocating fixed size caches on each server. Thus by itself, memcached is not effective for making use of varying amounts of spare memory.

Our goal in Mortar is to expose unallocated system memory so that applications such as memcached can make better use of it. By having the hypervisor control access to this volatile storage area, Mortar can prioritize how different guests access the memory and allows it to be reclaimed much more quickly than if it must be ballooned out of the guest. Mortar uses a modified Xen hypervisor that exposes a new hypercall interface for putting and retrieving data in the free memory pool. We believe that this interface will be useful for a wide range of scenarios at both the system and application level. In this paper we present two examples: a modified version of memcached that taps into spare hypervisor memory and an OS-level disk block prefetching system.

3. Hypervisor-Managed Resources

Traditionally, the guest OS and applications controlled resources that were statically assigned to the VM, and the hypervisor primarily provided isolation. While this offered the strictest performance guarantees, the overall resource utilization of the physical machine could be low because the hypervisor did not take different workload demands and spare resources into consideration.

To overcome this limitation, dynamic resource management [25] emerged to control the usage of memory according to priority and necessity. For instance, the balloon drivers in the hypervisor can monitor memory access patterns of each VM and grant more memory to more aggressive ones. This has many benefits by allowing VMs to grow and shrink based on their demand, but it can still cause some serious problems since the hypervisor does not know what memory is being used for and the VM does not know when it will get more or less RAM.

To show this, we run a memcached VM in an environment with automated memory ballooning. In the experiment, a memcached VM is first assigned all spare memory, but then

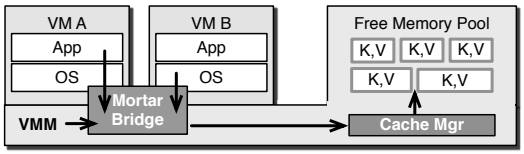


Figure 4: Mortar allows an application or OS to store Key-Value data in the hypervisor's free memory pool.

the allocation changes based on the memory trace shown in the upper part of Figure 3(a) (see Section 6.5 for the full experimental setup). As the memory allocated to the VM varies over time, pages from memcached must be swapped to disk, causing the response time to rise to several seconds, worse than not using the cache at all.

Not only is performance terrible due to swapping, the fact that data needs to be written to disk for each memory reconfiguration can dramatically increase the amount of time required for resource management operations. Figure 3(b) shows the time needed to reduce the memory allocation when using ballooning on a VM with resident data in memory. This can easily take tens of seconds if multiple gigabytes must be written to disk.

Both of these problems occur because of the *semantic gap* between the hypervisor and VM: the operating system and applications within the VM cannot distinguish between memory dedicated to the VM and memory which may soon be reclaimed by the hypervisor. In Mortar, we bridge this gap by not only allowing the hypervisor to dynamically allocate memory at fine granularity, but to understand how that memory is being used—it separates a VM's memory into that used for crucial application data and that used for volatile pages which can be recovered elsewhere if needed. While we focus on using this for disk (Guest OS level) and application level caches, our approach of granting the hypervisor greater knowledge and control of memory holds promise for a wide variety of general purposes.

4. Mortar Framework

The Mortar framework is divided into two main components as shown in Figure 4. The Mortar Bridge is composed of a pair of interfaces at the hypervisor and kernel levels that allow user applications to transfer data to and from the hypervisor's free memory pool. This interface can be accessed via system calls within user-space, or with direct hypercalls in kernel-space. The request to put or retrieve data from the hypervisor is passed to the Mortar Cache Manager, which is responsible for managing the hypervisor's free memory pool. This section describes the interface of the Mortar Bridge and how it is used by our two prototype applications. We then describe the eviction and management policies supported by Mortar in Section 5.

4.1 Repurposing Unallocated Memory

A hypervisor maintains a list of spare memory that can be allocated to VMs on demand. In order to use this spare mem-

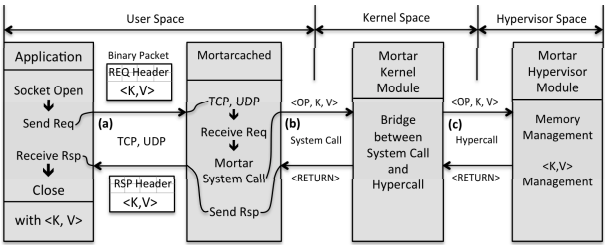


Figure 5: Mortar Protocol Processing Flow: (a) Equivalent protocol with memcached supports the same access method to Mortar so that we do not need to change applications; (b) System call moves data from user to kernel; (c) Hypercall bridges between kernel and hypervisor.

ory as a cache, we need a way to easily and quickly transfer data between a guest VM and the hypervisor controlled free memory pool. Mortar does this by defining a new Linux system call and a Xen hypercall which together provide the interface to a key-value store. Both of these calls take a key, a value, an operation (put, get, or invalidate), and an optional field that can set an expiration time for a new object. Communication between a VM and the hypervisor in Xen can be done through hypercalls, event channels plus shared memory, or the xenstore. Event channels plus shared memory and xenstore have limitations when transferring large amounts of data, whereas a hypercall delivers a physical address to the hypervisor, which can translate the physical address into the machine address and copy the data, so we use this approach. Depending on how Mortar is being used, these calls can originate in a user-space application or inside the guest VM's kernel; for this explanation we assume requests originate in user-space since this subsumes all the steps needed for the kernel case.

On a *put* operation, the calling application provides a key and value to the Mortar kernel module, which copies the data from user space to kernel space and invokes a hypercall. Moving objects from user space to hypervisor space via kernel space is necessary because no direct connection is possible from the user perspective. While the memory copies from user space to kernel space and then hypervisor space has a processing overhead, directly copying non-contiguous memory from user space is a non-trivial problem since the hypervisor does not know how the virtual address space is organized. If the hypervisor has enough unallocated pages to store the object, it is copied into the host's free memory. The pages used for the object are then moved from the hypervisor's free page list to a new "volatile page list", indicating that the page is being used to store cache data, but that it can be immediately reclaimed if necessary. A *get* operation reverses this procedure: the key is used as an index to a chained hash table which verifies the object is still in memory and copies it back to the kernel and then the user space application. Since the hypervisor is invoked on each operation it can verify the VM should have access rights to the data and can enforce prioritization across VMs.

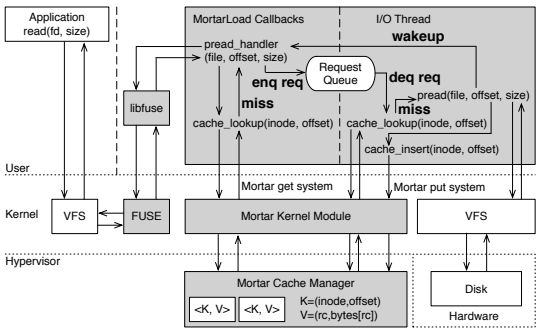


Figure 6: MortarLoad disk caching and prefetching.

4.2 Mortar-based Memcached

While Mortar's data store could be used for many purposes, our first prototype uses it to store data following the memcached architecture. We have modified the memcached application so that instead of using a fixed memory region to store all cached data, it invokes the Mortar system call to ask the hypervisor to hold the data. This modified memcached process can then be run in Dom-0 or a guest VM, and can be seamlessly merged with an existing memcached server pool. This lets Mortar instantly be used by many existing applications to access a distributed memory pool available throughout the data center.

Mortar modifies the backend memory management routines in memcached to change the course of the put and get functions so they route data to the hypervisor rather than user memory. Since the hypervisor may revoke memory storing an object without notifying memcached, a get request may return an error code for a missing object. Note that this is no different from what would happen in a regular memcached server if the object has been evicted, so we require no changes to existing applications.

Figure 5 shows the protocols of applications, Mortarcached (our modified memcached), kernel, and hypervisor. First, a web application issues a request to Mortarcached using the standard memcached protocol. Mortarcached receives the binary packet and checks the operation code. A get or put system call is then issued to the Mortar kernel module, then the kernel module simply delivers the request to hypervisor space by calling a new hypercall. Later in Section 6.2, we will show how much overhead occurs due to this additional processing. Modifying an application such as memcached to work with Mortar is a straightforward process (e.g., adding about 500 lines of code).

4.3 Mortar-based Prefetching

This section describes the design and implementation of our prefetching and caching system, MortarLoad. This system leverages the easy access to the free memory pool that is provided by Mortar to automatically prefetch data from storage systems (local or network disks) based on access predictions, and store the data in Mortar memory to expedite future accesses. MortarLoad is completely transparent to user

applications. At the highest level, an application requests I/O operations through standard read() and write() system calls, which will be forwarded to MortarLoad. Depending on where the data is, MortarLoad will fetch it from Mortar, or pass the request to underlying storage systems.

MortarLoad can be implemented in both kernel and user spaces. In this work for easy implementation, we implement a prototype of MortarLoad in Linux as a FUSE filesystem with backend calls through the Mortar hypervisor API. We leave a kernel implementation as future work and expect to achieve higher efficiency and performance. Figure 6 presents the overall architecture of MortarLoad.

MortarLoad adds an additional cache layer beyond the operating system's standard disk cache. This second-level cache uses spare memory provided by the Mortar framework. The cache management and replacement algorithms are managed by Mortar in the same way as memcached. In fact, requests from memcached and the disk can be stored simultaneously.

MortarLoad translates every I/O request into a tuple of <key, value>, where the key into the cache represents the inode of the requested file, file offset, and size of the operation, i.e., a function f(inode, size, offset). Requests are automatically aligned to 128KB-sized blocks by the FUSE layer.

For a read request, the Mortar cache is checked for the presence of this key. This call crosses the system call boundary into the kernel and then as a hypervisor call across the VM boundary. If the request can be satisfied from cache, a copy of the data is copied back from the hypervisor by Mortar. If the request is not in cache, it is enqueued for a separate I/O thread to handle. Serializing disk I/Os through a separate thread is to improve the performance when there are many simultaneous readers, especially when there are prefetch requests. This thread has an input request queue and an associated condition variable that it waits on.

Each request enqueued to the I/O thread also has a destination buffer and a blocking semaphore. When the I/O thread wakes up, it dequeues the latest request, performs an additional cache lookup (in case different threads requested the same block), and if not found, reads the data from the disk with a call to pread(). The resulting data and return code are copied to the location pointed to by the request and the associated semaphore increased. Another call is also made to place the key inode, size, offset and data result code, data bytes into the cache. When the calling context is woken up after waiting on the semaphore, it copies the data back into FUSE which then copies it to the application.

Prefetching is handled by having the calling thread place additional requests on the I/O queue that read several blocks ahead of the current request. The amount of prefetching is an adjustable start-up parameter. These requests have no waiting semaphore, but are still placed into the cache after being read from disk. Our experiments have shown that the prefetching accuracy is very high ($> 99\%$) for many workloads that perform sequential reads. As an enhancement, we plan to investigate the use of feedback-directed prefetching to vary the aggressiveness based on recent performance and the size of memory available to Mortar.

For a write request, MortarLoad currently puts the request straight into the I/O thread's input queue without any cache lookup. In other words, writes are handled with a simple write-through policy. If the I/O request is a write and is writing to an already-cached block, that block is first removed from the cache, and then written to disk normally. As a possible enhancement, we plan to investigate optimizations including a write-back cache.

There are other filesystem operations which as currently implemented do not make use of the cache at all, to name a few, getattr, access, readdir, chmod, chown, and fsyn. Instead, these operations write directly to the disk, bypassing the I/O thread, and invalidate the corresponding cache entries as needed.

5. Cache Management Mechanisms

Mortar's cache management has two important roles: (1) handling data replacement/eviction; (2) enforcing VM priorities based on weights.

5.1 Cache Replacement Algorithms

Mortar uses inactive memory to store application data, but it is possible that this memory will suddenly be needed for either a new, migrated, or overloaded VM. Fortunately, since the data store is considered volatile, Mortar can invalidate cache entries without needing to worry about consistency. Ideally, cache eviction should follow an intelligent policy such as removing the least recently used (LRU) entries first, however, this can be too slow if gigabytes must be freed and each cache entry is on the order of kilobytes.

The Xen hypervisor uses Two-Level Segregate Fit (TLSF) [17], which is a general purpose dynamic memory allocator specifically designed to meet real-time bounded response times. The worst-case execution time of TLSF memory allocation and deallocation has to be known in advance and be independent of application data. With this, Mortar must support two different memory release schemes: a slower, but more intelligent scheme used to replace objects when the cache is full or when only a relatively small amount of memory needs to be freed, and a fast release approach that can quickly purge a large portion of memory. This allows efficient cache management in the normal case, but still allows memory to be rapidly reclaimed when needed for other purposes.

Slow Cache Replacement Algorithm (SCRA): We use a hybrid cache replacement algorithm, which prefers to evict expired objects, but falls back to a combination of LRU and least frequently used (LFU), called LRFU, to improve the combined results [12]. LRU requires keeping age records

for caches, and LFU needs to keep reference counts; Mortar tracks this information on a per-object basis, and also indexes objects by VM in order to support the cache prioritization scheme described in the next section. The algorithm works by alternating between removing the least recently used item or the least frequently used one. The combination of LRU and LFU tries to balance the drawbacks of each: unpopular objects that happen to have been accessed recently may still be evicted, and content that was briefly popular some time in the past may be removed if it has not been touched recently. Mortar uses SCRA when replacing objects in the cache, or when a relatively small amount of memory (e.g., up to 1GB) must be freed for other uses.

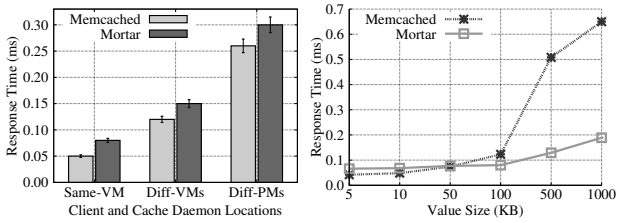
Fast Cache Replacement Algorithm (FCRA): Since Mortar tries to fill all the spare memory in the system, it must be prepared for the situation when the system needs to free a large amount of memory instantly. Dynamic resource management techniques may require additional RAM to be allocated to an important VM, and since the cloud service model is pay-as-you-go, users may turn off and turn on their VMs frequently, causing sudden demands for large amounts of memory. In these scenarios, Mortar must guarantee fast cache eviction to prevent delays in resource management operations.

Mortar's fast cache eviction algorithm works by simply stepping through the hash-chain used to store all of the object keys, removing them in order. Since the hash function essentially randomizes the object keys, this results in a random eviction policy. Since no cache frequency or recency information needs to be used or updated, this can be performed very quickly. While FCRA allows large numbers of objects to be removed from the cache in a short period of time, it may harm the performance of applications since hot data may be inadvertently evicted from the cache.

5.2 Weight-based Fair Share Algorithm

Mortar uses a weight-based prioritization system to determine how cache space is divided when multiple VMs compete for cache memory. If one VM is assigned twice the weight of another, then the higher weight VM will be allocated twice as much cache space. However, if a high weight VM does not use its entire allocation, a lower weight VM will be able to fill the spare capacity with its own data. If the high weight VM later needs more storage space, the lower priority VM's data will be evicted.

Mortar's weight-based proportional fair partitioning scheme works as follows. Let $W = \{w_1, w_2, ..., w_N\}$ be a set of weights and $C = \{c_1, c_2, ..., c_N\}$ be a set of current cache utilizations, where w_i and c_i are the weight and current cache use of VM i, and N is the number of VMs. We denote P as the total cache capacity. The weight ratio of VM i is $r_i = \frac{w_i}{\sum_{j=1}^{N} w_j}$, and the fairness parameter is $f_i = \frac{c_i/P}{r_i}$. If a virtual machine has $f_i > 1$, this indicates that it is using more than its weighted fair share of the cache.



(a) Response Time Overheads (b) Mortar Value Size Benefits

Figure 7: (a) The overhead of Mortar is on the order of 0.03ms compared to memcached; (b) Mortar has better performance over memcached when the value size becomes larger than 50KB.

When the cache is fully utilized, the objective is to ensure:

$$f_1 = f_2 = ... = f_N. \quad (1)$$

Equation (1) divides the cache size proportionally based on the N virtual machines' weights. This is achieved by Mortar's Cache Manager with consideration of the fairness metrics when handling a put request. If there is spare capacity in the cache, then Mortar will always allow a VM to add the object, regardless of its current fairness value. However, when there is no cache space left, Mortar finds the VM with the largest fairness metric f and evicts data stored by that VM in order to fit the new object. This ensures that VMs unfairly utilizing excess capacity must release their data when another VM wishes to use its weight-based allocation.

6. Experimental Evaluation

Our goals for the evaluation is to see the overheads of Mortar through micro-benchmarks, and to check the performance for both Mortar-based memcached and prefetching through real workload-based benchmarks.

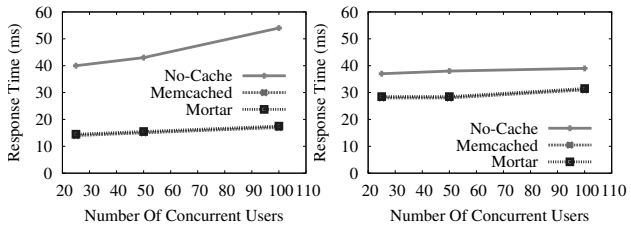
6.1 Environmental Setup

System Setup: Six experimental servers, each of which has quad-core Intel Xeon X3450 2.67GHz processor, 16GB memory, and a 500GB 7200RPM hard drive. Dom-0 is deployed with Xen 4.1.2 and Linux kernel 3.5.0-17-generic, and the VMs use Linux kernel 3.3.1.

Memslap[1] **(micro-benchmark):** For our micro-benchmark experiments, memslap from a memcached client library is used. It generates a load against a cluster of memcached servers with configuration options including number of concurrent users, operation type, and number of calls.

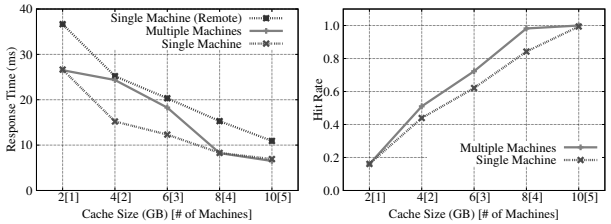
CloudStone Benchmark [21]: CloudStone is a multi-platform benchmark for Web 2.0 and Cloud Computing. It is composed of a load injection framework called Faban (client), and a social online calendar Web application called Olio (server). CloudStone provides a framework to generate

[1] http://www.libmemcached.org



(a) Home Page (b) Tag Search

Figure 8: When combined with a realistic web application, Mortar's overheads are insignificant.



(a) Response Time (b) Hit Rate

Figure 9: Increasing the total cache size (either on a single machine, or divided across multiple), increases hit rate and reduces respones time when Mortar is accessed with a Zipf (α = 0.8) request distribution.

workloads of varying strengths and measure application performance. The PHP-based Olio web application queries a memcached node before issuing read requests to its MySQL database.

6.2 Mortar Overheads

Mortar keeps data in hypervisor space, but allows it to be accessed from both kernel modules and user space applications. To evaluate the overhead of Mortar's operations, Figure 7(a) shows the time to perform a *get* request using both standard memcached and our Mortar-based version. Kernel-based applications that make use of Mortar's data store should see a lower level of overhead since data will not need to be moved to user space. Since Mortar requires two data copies for each request, it incurs a higher overhead than a traditional memcached server, which uses only preallocated user space memory.

We test each system using the memslap benchmark, and report the average response time and standard deviation for 100 requests, each of which has a 100B key and 5KB value. We test three scenarios: 1) when the memslap client is in the same VM as the cache daemon, 2) when in a different VM but the same host, and 3) when the client is on an entirely different physical machine (the most common case in practice). In each case, we find that the overhead of Mortar is quite small, on the order of 0.03ms, less than 15% overhead if the cache must be accessed over the network. We believe this overhead is a small price to pay in exchange for opening up a larger amount of memory for the cache. Of course, our approach can be used in conjunction with regular memcached servers, allowing for fast, guaranteed access for priority applications and slower, best-effort service to applications using the Mortar memory pool.

We next consider how the data value size affects Mortar's overhead. Figure 7(b) shows that data size does not have a significant impact on Mortar's response time. Memcached is designed primarily for web applications that must store relatively small objects (maximum size $\leq$ 1MB), but Mortar is a general data storage framework, so we specifically use a memory allocator, TLSF, that supports a more consistent memory (de)allocation speed regardless of size.

6.3 Web App. Performance Overheads

The previous experiments show the low-level overheads of Mortar, but it is also important to see how it performs with a more realistic web application. We use the CloudStone benchmark [21] to measure how Mortar's overheads affect the performance of a real application. We dedicate an identical amount of memory to both Mortar and regular memcached and measure the client performance under a range of workloads.

Figure 8 shows the performance of CloudStone when 25 to 100 concurrent users connect to Olio. We consider the four most common operation types since together they make up over 95% of all requests; the other request types perform similarly. The results show that the Home Page (Figure 8(a)) operation has the biggest difference between no-cache and cache because they involve many database accesses to a small set of hot content. We find that Mortar and memcached have essentially no difference in application-level response time, despite the minor overheads shown by Mortar when handling small requests in the previous section.

The Tag Search operation shown in Figures 8(b) has similar performance. Since these operations access a much wider range of database records, it is less likely that requests will be found in cache, reducing the overall performance benefit compared to the no-cache case. Once again, we find that Mortar incurs no overhead compared to a standard memcached deployment.

In all of the tests, we find that the performance for Mortar and standard memcached scales identically as the number of clients rises. This suggests that our Mortar-enhanced version of memcached has both minimal additional latency and can support a similar level of concurrency as traditional memcached.

6.4 Impact of Cache Size

The goal of Mortar is to opportunistically make use of all free memory, so we next consider how application performance varies with the size of cache available. We use a simple web application that maintains a database filled with 10GB worth of entries, each sized at 50KB. We vary the size of the cache and measure Mortar's response time and hit rate.

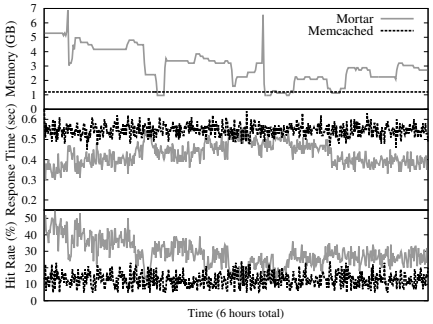


Figure 10: Mortar makes use of all spare memory, leading to lower response times and a higher hit rate than memcached.

We allow the cache to warm up to a consistent hit rate after each size changes.

To demonstrate the benefits of more cache space, Figure 9 shows how cache performance changes while varying the available memory both in a single machine (accessed either locally or by a remote client) and for multiple machines (one local and up to four remotes, each adding 2GB of cache space). As expected, performance improves as the size rises.

6.5 Dynamic Cache Sizing

To truly see the benefits of using Mortar, we need to consider a scenario where the amount of memory available for the cache varies over time—a scenario which memcached is unable to take advantage of since the cache is statically sized. To demonstrate this, we take one of the memory traces from our IT department shown in Figure 1 and condense it down to a six hour period. We compare two cases: 1) a traditional memcached server with a fixed cache size of 1.2 GB (the largest a fixed sized cache can be over the entire trace) and 2) our Mortar implementation that can scale the cache size up and down based on the server's free memory. We use the cache as a frontend to a MySQL database that is filled with 16GB of data, with an average record size of 50KB. When memory needs to be reclaimed from the cache, we use Mortar's slower, but more accurate, eviction policy; as will be shown in Section 6.8, this still allows memory to be freed in under one second.

Figure 10 illustrates the memory available to each cache, the response time, and the hit rate as web requests are processed over the course of the experiment. The clients make requests at a constant rate, but queries follow a Zipf distribution with $\alpha = 0.8$, resulting in the type of skewed distribution commonly seen by web applications that have a relatively small portion of more popular content [29]. Since memcached has a fixed size cache throughout the trace, its performance is relatively steady with an average response time of 0.57 seconds and a hit rate consistently below 20%. In contrast, Mortar's performance varies based on the amount of available cache space, with a response time ranging from 0.3 to 0.52 seconds. Overall, Mortar has an average response time of 0.38 seconds, a 35% improvement over memcached.

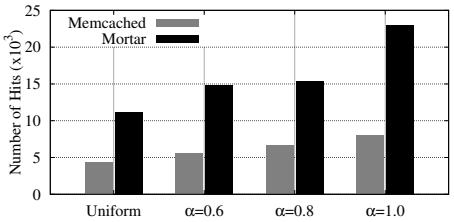


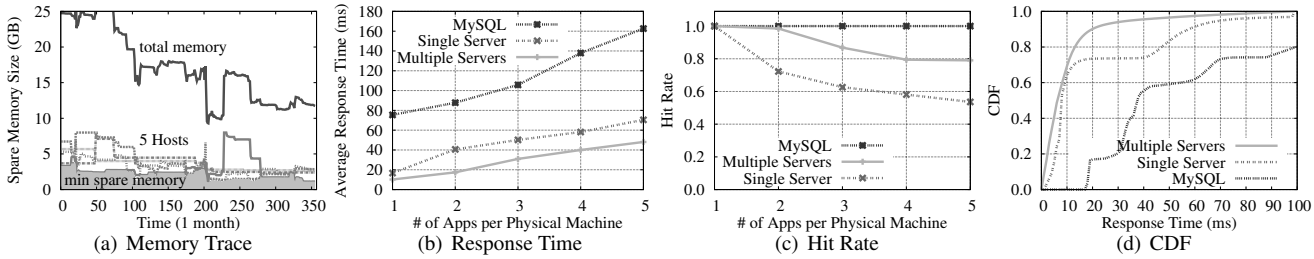
Figure 11: Varying the request distribution affects the likelihood of data being within the cache. In all cases, Mortar has substantially more cache hits since it has more memory available to it.

We next study the impact of the request distribution on cache performance. Figure 11 shows the total number of cache hits when changing the request distribution for the experiment described above. With a uniform distribution or a Zipf distribution with a low α value, it is less likely that requested content will have been seen recently enough to still be in the cache. In all cases, Mortar provides a substantial benefit over memcached since it is able to make use of about 2.6 times as much cache memory over the course of the experiment.

6.6 Multi-Server Caching

The previous experiment illustrates the benefits of Mortar, but also shows how the variability in free memory on the caching server can result in less predictable performance. To mitigate this drawback, we next experiment with Mortar in a larger scale setting where multiple hosts each run both applications and a Mortar cache. We use a set of five memory traces from our university's data center, which have been scaled down to prevent the free memory on each host from becoming over half of the total memory size, as shown in Figure 12(a). The total spare memory is initially close to 25GB, but goes through several changes before ending around 13GB. No host has memory size below 1.2GB (min memory).

Our goal is to understand how having a larger number of servers available for caching data can reduce the performance variability of the applications using the cache. Towards this end, we compare two setups: the *single-server* case where a single server acts as a cache, and the *multi-server* case where five servers use their combined spare memory for the cache. In each case we vary the number of applications active *on each physical host* from one to five. For the single-server case we select the trace that on average has the most free memory available, and this is used to cache data by up to five applications. In the multi-server case, up to twenty-five applications distribute their data across all five hosts. If the data cannot be stored in the cache it must be retrieved from a MySQL database.



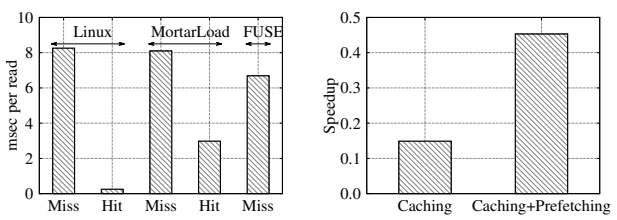
(a) Memory Trace (b) Response Time (c) Hit Rate (d) CDF

Figure 12: Running Mortar on more servers reduces performance variability; (a) depicts the system memory traces over time; (b) shows the average response times; (c) illustrates the hit rates; (d) cumulative density functions (CDFs) for response time.

Figures 12(b) and 12(c) illustrate the average response time and the hit rate over the entire memory trace. Even though the multiple server scenario has a larger number of total applications running, it has both a 20ms better response time and 20% better hit rate than the single server setting. This happens because the variations in free memory on each of the five hosts do not generally occur at the same time, increasing the chance that at least some application data will be found in the cache compared to the single server case where periods of memory scarcity significantly impact all applications. Figure 12(d) depicts the distribution on response times, and further reaffirms the result that spreading Mortar's data across multiple servers leads to not only improved response times, but a lower standard deviation.

6.7 Disk Prefetching with Mortar

We next evaluate the overheads and benefits of MortarLoad using the following I/O benchmarks: **Slowcat** is a synthetic benchmark written by us to compute typical I/O times in a carefully controlled manner. This program will read in a file in increments of a configurable block size and sleep for a configurable amount of time per read. It reports several usage statistics, including the total time spent waiting for read operations to complete. **Videoserver** is part of the Filebench suite of I/O benchmarks [8]. It is intended to mimic the behavior of a streaming video server that reads and writes to several video files at a time. In our configuration, we set it up for reading three videos and writing one. The overall I/O amount is approximately 8GB read and 1.5GB written. **OLTP** is the database application benchmark in Filebench, which has a significant percentage of non-sequential reads.



(a) Elapsed Time for Single Reads (b) Videoserver Speedup

Figure 13: Caching + Prefetching Experiments

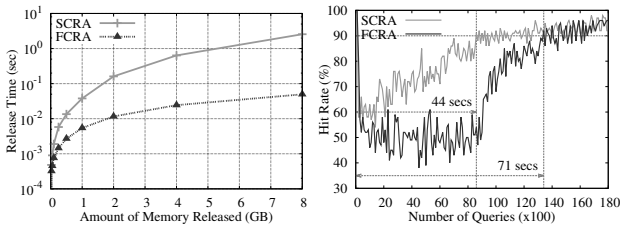
MortarLoad Overheads: We first study the overheads of our FUSE based MortarLoad when prefetching is turned off. Figure 13(a) shows the average read time in milliseconds measured with slowcat for a 1GB input file read with 1 MB block sizes under different conditions. We first measure the uncached base Linux time by forcibly dropping caches, and executing the program. Once this completes, we re-run it to measure the time taken when the entire file fits in the Linux page cache. To measure the impact of MortarLoad, we mount the MortarLoad file system to act like a caching loop back file system with a Mortar cache size greater than the input file size. We perform a similar measurement, this time flushing the Mortar cache (and Linux cache) before the first pass and re-running for a second pass (with just Linux caches dropped) over the Mortar cached data. The final bar measures the cache miss cost on vanilla FUSE, which is known to actually improve performance for certain cases, which turns out to be the case for this particular workload.

Our results show that MortarLoad has similar performance to Linux when they experience a cache miss and must read from disk. A MortarLoad cache hit is slower than a standard Linux cache hit but in normal operation, the Linux page cache acts as a first-level cache, only reading from Mortar when it fails to hit in the page cache. This means there is not a performance reduction to regular cache hits from using MortarLoad, but that a second-level cache hit to the prefetched data is somewhat slower than the base cache. We expect that the MortarLoad cache hit latency can be reduced by further optimizing our implementation and moving it into the kernel.

	Cache Size (MB)		
	100	500	1000
Avg. Read Latency Speedup	2.84	3.14	3.79
Avg. Hit Percentage	4.51	22.06	59.72

Table 1: Data prefetching with different memory sizes running the OLTP benchmark.

Cache and Prefetching Benefits: We next study how MortarLoad can improve the performance of the Videoserver benchmark by providing both a larger cache and prefetching. Figure 13(b) shows the overall speedup with and without prefetching compared to a baseline without FUSE. These



(a) Release Time (b) Recovery Time

Figure 14: (a) FCRA releases memory an order of magnitude faster than SCRA. (b) However, the speed of FCRA has the cost of reducing cache performance for a longer period.

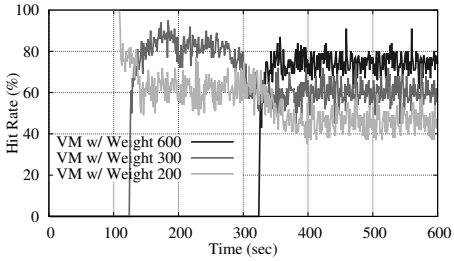


Figure 15: Three VMs with weights 200, 300, and 600 run web applications starting at times 0, 100, and 300, respectively.

results are measured using an initially empty 1GB cache. The benefits of using prefetching are substantial over using a plain LRU disk cache (speedup of 45% compared to 15%).

We next evaluate the performance of the OLTP server while we vary the size of Mortar cache. As shown in Table 1, with a 100MB cache, the server sees 2.8× speedup on average read latency when compared to the case of native read. When the cache size is set to 1GB, the hit rate is further increased to 60% and as the result the read improvement reaches 3.8×.

Our results illustrate how re-purposing spare memory for an extended disk cache with prefetching provides substantial performance improvements. Further, Mortar transitions the management of memory from inside each VM's OS to the hypervisor, which allows for higher-level decision making as shown in the following sections.

6.8 Responding to Memory Pressure

Mortar's goal is to opportunistically use all free memory, but it must be ready to release memory for various situations: when a new VM enters the system, when a VM in another physical machine live migrates to this host, or when a VM's memory allocation needs to be adjusted.

In this experiment, we measure how much time it takes to release the demanded amount of memory when the cache is filled with 50KB objects. We compare the two cache replacement algorithms, SCRA (slow but accurate) and FCRA (fast but random), discussed in Section 5.1. Figure 14(a) shows the release time as the amount of memory requested increases. As expected, both approaches take more time for larger requests, but the FCRA approach can keep the release time under 0.1 seconds even when releasing the full 8GB cache (approximately 168 thousand objects), while SCRA takes more than ten times as long.

Depending on the speed with which memory requests must be handled, Mortar can be tuned to select which replacement algorithm is used for different size requests. In many systems, a two second latency for memory requests is acceptable, meaning that SCRA can be used exclusively, increasing the likelihood that hot data will remain in cache. Other systems may not be able to tolerate this latency, so, for example, a 1GB threshold might be used to switch between SCRA and FCRA. This would allow all memory reallocation requests to be handled within only 0.1 seconds.

While FCRA is clearly faster at relinquishing memory, this comes at a cost since it removes objects without considering cache locality. To study how the eviction algorithm affects cache performance, we next test a scenario where the cache is rapidly resized, and then the time it takes to rewarm the cache is measured. We start the experiment with a hot cache filled with 15GB of data. We then cut the cache size by 8GB, causing more than half of the data to be evicted by one of our two replacement algorithms. We then immediately increase the Mortar memory pool back to 15GB and observe the time required to recover the previous hit rate.

Figure 14(b) shows how the hit rate of the FCRA and SCRA managed caches recover over time. During the mass eviction, SCRA is able to preserve a larger amount of hot data by using frequency and recency data, so its initial hit rate is higher than FCRA which randomly removes data from the cache. This gives SCRA a significant edge on rebuilding its cache, allowing it to reach a 90% hit rate 39 percent faster than FCRA.

6.9 Weight-based Memory Fairness

Mortar supports weight-based proportional fair partitioning to divide the cache between competing VMs. VM's receive cache space proportional to their weight, but if there is spare capacity, even a lower weight VM can make use of it. Figure 15 shows hit rates of three VMs when they are assigned different weights. Each VM uses web server-type applications with request arrival rates following Zipf distribution ($\alpha = 0.8$). Each VM starts at a different time, causing the relative weights to adjust over the course of the experiment. A VM with weight 200 (VM_200) starts at time 0, and uses the whole cache because no other VMs are active. At time 100, a VM with weight 300 (VM_300) starts pushing data into the cache, so VM_200 surrenders cache space to VM_300 according to Equation (1). At time 300, a VM with weight 600 (VM_600) starts causing the cache to be rebalanced again.

While Mortar is able to correctly reallocate cache space to each of the VMs based on their weights (e.g., VM_600 receives $\frac{6}{11}$ of the cache), the same proportion does not necessarily hold for the hit-rate or response time achieved

by each VM. This is because in a skewed distribution like Zipf a smaller cache may still fit the most important hot data. This illustrates one of the challenges in partitioning a shared cache such as Mortar: weights can be used to control resource shares, but they may not be directly proportional to performance.

7. Related Work

Our system draws inspiration from the Transcendent Memory (tmem) system [16]. Tmem is made up of a set of pools in the hypervisor that can be used to store the disk cache pages of each VM. Tmem provides an efficient way to manage the cache by providing functions such as compression and remote cache access. Mortar provides a general purpose data store, while tmem focuses on swapping the ownership of full memory pages between the guest and hypervisor to facilitate disk cache management.

Dynamic memory management systems automatically control memory allocations for multiple VMs, typically by using a "ballooning" mechanism to add and subtract pages from the guest [25, 28]. Waldspurger [25] tracks individual VM memory usage by monitoring page access rates. This allows it to grow and shrink VM allocations as needed. Zhao et al. [28] look at multiple VMs at once and decide how to divide up memory. Perfectly allocating memory is impossible since workloads may change over time, so even when using these systems system admins often leave spare memory to handle new or rising workloads. The Overdriver system proposes using network based RAM in times of high load [27]. In contrast, Mortar focuses on opportunistically using memory during periods of light load.

Page sharing schemes such as transparent page sharing (TPS) [10, 25] have been proposed to maximize memory efficiency. TPS provides a level of abstraction over physical memory and is able to share pages by identifying identical content. TPS is another way of freeing up a moderate amount of memory, but this memory often does not last for long periods of time, as shown by the Satori [18] system. We believe that Mortar can be used very effectively in conjunction with TPS systems by allowing even small numbers of briefly shared pages to be put to an effective purpose.

Data prefetching has been extensively studied on CPU cache [22], hard drives [7, 9], and most recently, solid-state drives [14]. Another very related topic, disk caching, also draws a great amount of interests for improved performance, as well as energy efficiency, and here we cannot possibly present a complete list. To name a few, [2, 5] propose a gray-box approach to infer and utilize OS and file caches, and [6] present work combining the study of both.

Smart caching mechanisms in the hypervisor have also been proposed, for example, [26] combines all persistent storage in a virtualized cluster and uses local persistent storage as cache to VMs, [13] infers disk block liveness to manage VMM memory cache, and [15] describes a caching policy split between the hypervisor and guest VMs. While [1] argues lack of disk locality for certain workloads, it suggests to use memory as a cache in data centers based on the observation of memory locality. MortarLoad focuses on data prefetching for file-level accesses within each guest VM, and is able to leverage the free memory pool managed by Mortar, which is otherwise unavailable to the guest VMs.

8. Discussion

Performance Unpredictability: To be effective, Mortar needs to have some amount of spare memory available throughout the data center. While the traces from our IT department and anecdotal discussions with system administrators indicate that this is commonly the case, a valid concern is that Mortar will lead to deceptively high performance when workloads are light, and poor performance when workloads rise and there is less spare capacity. Of course one solution is to use Mortar as a supplemental cache in addition to a set of regular memached nodes. In practice however, if Mortar is deployed in a cloud-scale data center we do not expect this to be a major concern since different applications will see workload spikes at different times.

Security: Memory and cache-induced side channel attacks are a concern in shared environments [20]. To increase security guarantees, Mortar allows to choose whether to put objects in shared memory (potentially reducing redundancy among trusting VMs) or private memory. This separation does not affect how memory is managed, but allows Mortar to control permission for each data access. While this prevents data leakage, Mortar does potentially expose information about the amount of spare capacity on the host, and in turn the memory utilization level of co-located VMs. Still, the random behavior of VMs and unknown memory partitions make hard to infer the real metrics.

Private Clouds: Within a private data center, system administrators can use Mortar both for exploiting spare capacity and as a way to gain finer control over memory allocations. Mortar moves memory management from within each VM's operating system down into the hypervisor, which may have more information about the relative priority of different virtual machines. If an important application is known to require a memcached size ranging from 100MB to 1GB depending on its workload, then the administrator can assign 1GB of memory to Mortar, and mark the application with a high priority so that it will be able to get the cache space it needs when its workload is high, but allow another VM to use the spare memory (perhaps for disk prefetching), when the workload is low. This kind of flexible, fine grained memory management is impractical with existing memory ballooning techniques.

Public Clouds: We envision that a public cloud may use Mortar to pool together its spare memory resources and sell them at a discounted price. For example, Amazon's spot-instance market auctions off spare resource capacity in the form of VMs which may be instantly shutdown if the

provider needs those resources for a higher paying customer. Mortar allows a similar approach to be used for memory at a very fine grain. The popularity of the spot market illustrates that developers will eagerly make use of even highly transient resource capacity if the price is right.

9. Conclusion

Mortar represents the start of our vision for new techniques that opportunistically consume idle resources in a data center without imposing overheads on other active applications. It does this by taking the unallocated memory on each server and exposing it as a volatile data store that can be rapidly reclaimed by the hypervisor if needed. Our prototype modifies the Xen hypervisor to expose this interface to a memcached server and a disk prefetcher. This allows existing web applications and the OS to immediately make use of memory that currently is left idle as a buffer for rising workloads. Mortar moves the control of memory from within each VM's operating system to the hypervisor level, and allows it to be managed at finer granularity than existing approaches that rely on resizing VM memory allocations. In our future work we will investigate further uses of the Mortar framework, as well as how shifting the control of resources from inside a VM's operating system to the hypervisor can allow cloud platforms to make smarter management decisions.

Acknowledgments

We thank the reviewers for their help improving this paper. This work was supported in part by NSF grants CNS-1253575, CNS-1350766, and OCI-0937875.

References

[1] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 43–56, Banff, Alberta, Canada, 2001. ACM.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *In Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[4] L. A. Barroso and U. Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009.

[5] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box knowledge of Buffer-Cache management. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 29–44, 2002.

[6] A. R. Butt, C. Gniady, and Y. C. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 157–168, New York, NY, USA, 2005.

[7] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. Diskseen: exploiting disk layout and access history to enhance i/o prefetch. In *USENIX Annual Technical Conference*, pages 20:1–20:14, 2007.

[8] FileBench. http://sourceforge.net/projects/filebench/.

[9] B. S. Gill and D. S. Modha. SARC: sequential prefetching in adaptive replacement cache. In *Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA, 2005.

[10] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *USENIX*, 2008.

[11] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. *USENIX Internation Conference on Autonomic Computing*, 2013.

[12] P. R. Jelenkovic and A. Radovanovic. Optimizing lru caching for variable document sizes. *Combinatorics, Probability & Computing*, 13(4-5):627–643, 2004.

[13] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th international conference on architectural support for programming languages and operating systems*, pages 14–24, 2006.

[14] Y. Joo, J. Ryu, S. Park, and K. Shin. FAST: quick application launch on solid-state drives. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, pages 19–19. USENIX Association, 2011.

[15] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 3:1–3:15, Berkeley, CA, USA, 2007.

[16] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Transcendent memory and linux. *Oracle Corp.*, 2009.

[17] M. Masmano, I. Ripoll, A. Crespo, and J. Real. Tlsf: A new dynamic memory allocator for real-time systems. *ECRTS*, 2004.

[18] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. *USENIX*, 2009.

[19] 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. *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[20] R. Owens and W. Wang. Non-interactive os fingerprinting through memory de-duplication technique in virtual machines. In *Proceedings of the 30th IEEE International Performance Computing and Communications Conference*, PCCC '11, pages 1–8. IEEE Computer Society, 2011.

[21] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[22] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, 2007.

[23] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently meeting very strict, low-latency slos. *USENIX ICAC*, 2013.

[24] VMware. Resource management with vmware drs. *Technical Resource Center*, 2006.

[25] C. A. Waldspurger. Memory resource management in vmware esx server. *OSDI*, 2002.

[26] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: managing storage for a million machines. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, pages 4–4, Berkeley, CA, USA, 2005.

[27] D. Williams, H. Jamjoom, Y.-H. Liu, and H. Weatherspoon. Overdriver: handling memory overload in an oversubscribed cloud. In *7th International Conference on Virtual Execution Environments*, pages 205–216. ACM, 2011.

[28] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. *VEE*, 2009.

[29] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving cache by using less cache. *HotCloud*, 2012.