UniCache: Hypervisor Managed Data Storage in RAM and Flash

Jinho Hwang[†], Wei Zhang[*], Ron C. Chiang, Timothy Wood, and H. Howie Huang
[†]*IBM T.J. Watson Research Center* [*]*Beihang University* *The George Washington University*
Email: jinho@us.ibm.com, zhangwei1984@gwu.edu, {rclc, timwood, howie}@gwu.edu

***Abstract*—Application and OS-level caches are crucial for hiding I/O latency and improving application performance. However, caches are designed to greedily consume memory, which can cause memory-hogging problems in a virtualized data centers since the hypervisor cannot tell for what a virtual machine uses its memory. A group of virtual machines may contain a wide range of caches: database query pools, memcached key-value stores, disk caches, etc., each of which would like as much memory as possible. The relative importance of these caches can vary significantly, yet system administrators currently have no easy way to dynamically manage the resources assigned to a range of virtual machine data caches in a unified way. To improve this situation, we have developed UniCache, a system that provides a hypervisor managed volatile data store that can cache data either in hypervisor controlled main memory (hot data) or on Flash based storage (cold data). We propose a two-level cache management system that uses a combination of recency information, object size, and a prediction of the cost to recover an object to guide its eviction algorithm. We have built a prototype of UniCache using Xen, and have evaluated its effectiveness in a shared environment where multiple virtual machines compete for storage resources.**

***Keywords*-Cloud Computing; Memory Management; SSD**

I. Introduction

Applications and operating systems have many opportunities to improve I/O performance by caching data in memory. Operating systems aggressively use buffer and page caches to store data that would otherwise have to be retrieved from disk at much longer latency. Likewise, application-level caches such as memcached [14], [19] are used to store data such as the results of expensive to compute database queries. Applications and operating systems are eager to make this trade of memory consumption for performance because the cache can often be shrunk if memory is needed for another purpose.

Unfortunately, this is not necessarily the case in a virtual setting where multiple virtual machines (VMs) may aggressively consume whatever memory they are allocated for a cache [13]. The hypervisor (a.k.a. VMM – virtual machine manager) is unaware of the distinction between volatile data pages that can easily be recovered, and those which hold critical application or OS state. Even worse, the system memory that is not assigned is wasted due to the hypervisor's inability to flexibly manage spare memory. This lack of information and manageability prevents the hypervisor from efficiently managing memory resources since each VM appears to be actively using all of its memory, while in fact some of it may be able to be reclaimed without a significant impact to performance. Also, the system spare memory is not used efficiently.

We have developed UniCache to increase the hypervisor's control over the storage hierarchy. UniCache allows applications and operating systems to make calls into the hypervisor to access a key-value store spread across DRAM and Flash-based memory, e.g., a set of solid state drives (SSD) configured as a RAID array [2], [16], [21]. Here we combine a main memory cache with the new Flash memory layer to provide larger memory capacity and fast access [3], [7]. This requires UniCache to carefully allocate memory and SSD resources to a set of competing VMs with volatile data to store.

One of the insights in our work is that not all volatile data caches are equivalent, nor are the objects stored inside of them. For example, traditional disk caches employ least recently used (LRU) information to guide eviction [8], [12], [22]. This makes the assumption that the cost of bringing any object back into the cache is roughly the same, yet this is not the case for application-level caches such as memcached, where some objects may represent the results of very long running queries while others can be trivially recomputed. UniCache is able to predict the cost of recovering an object by tracking previous get/put request pairs. To efficiently support a wide range of cache types, UniCache uses an eviction policy that can weight the importance of recency information, object recovery cost, and object size.

UniCache's focus is on determining where data should be stored in a cache that spans multiple levels of the storage hierarchy, particularly when these storage areas need to be partitioned for multiple competing VMs. Our work on UniCache offers three primary contributions:

- An interface that allows applications and operating systems to store data across a storage hierarchy,
- A cache eviction algorithm that accounts for both temporal locality and data-specific features such as cost to recompute and size, and
- A cache partitioning scheme that estimates the performance benefit provided by the cache to each VM without requiring any application-specific data.

We have built a prototype system that can support a variety of uses such as memcached for web applications and an OS-based disk prefetching system. Our evaluation

of UniCache shows that a certain web application can provide a significant improvement−32% with memory and an additional 28% with SSD support.

Our paper is structured as follows: Section 2 provides the background and motivation for our work, and Section 3 provides an overview of our system architecture and the interface between applications and the hypervisor managed caches. We then discuss cache replacement and partitioning algorithms in Section 4. Section 5 provides our evaluation using several realistic data center benchmarks. We discuss related work in Section 6, and conclude in Section 7.

II. Background and Motivation

UniCache is motivated by two converging challenges: the growing importance of cached data for meeting performance goals and the increasing density of Cloud virtual environments where resources are multiplexed for multiple users. Single purpose caching solutions such as OS disk caches and database query caches have long been employed, but general purpose caches such as Memcached have seen growing popularity for maintaining web application performance; for example, Facebook is said to run more than 10,000 memcached servers [20]. Caching has become popular enough that several cloud platforms such as Amazon AWS and Windows Azure now offer "cache-as-a-service" products [1], [6].

Caches come in two varieties: dedicated deployments such as memcached where each node is allocated a fixed amount of memory, and opportunistic caches such as the Linux buffer and page caches that expand to consume underutilized memory [24], [27]. However, the use of virtual environments complicates and provides new opportunities for both of these approaches. An opportunistic cache within one VM may greedily consume memory pages for itself if no other process inside the VM is using the memory, but it is possible that a different, potentially higher priority VM on the same host could make better use of that memory. Similarly, dedicating fixed size memory regions to a memcached node can be convenient for offering a predictable quality of service level, but if spare memory is available on the host (either owned by the hypervisor or a lightly loaded VM), then why not allow the memcached process to expand into that memory space?

As data are usually divided into two categories: hot (popular) data and cold (unpopular) data [15], putting all data in the cache is certainly not efficient in terms of resource management and operating expenses. Still, since user experience is becoming one of the most important metrics, enterprises strive to reduce the end-to-end response time by having as much data cached as possible. Recent and upcoming storage advances such as flash memory, NVRAM [4], and Memresistor [10] promise dramatically larger capacity than DRAM, while providing much faster speeds than traditional spinning hard drives. Solid state drives have fallen in price and increased in capacity, but they still remain something of a luxury for the low cost servers commonly used in data centers. This is particularly the case for virtualized servers, where dozens of VMs may be competing for the host's resources. In this environment, dedicating an SSD per VM (or per cache) is far too expensive. Instead, it is desirable to have a memory hierarchy composed of DRAM and SSDs to be jointly managed by the hypervisor because it has more information about the relative priority of different VMs, as well as the full system resource availability.

UniCache allows flexible allocations of storage resources to operating systems and applications that wish to cache data by offering a unified caching service at the hypervisor level. Data is split between hypervisor controlled main memory and Flash memory to provide varying levels of performance based on application behavior and VM priority. Expanding the cache to include both memory and SSDs allows for a much larger amount of data to be stored at lower cost, which is very important for virtualized environments when competing VMs want to make use of limited memory resources.

By combining many diverse caches into one, UniCache must deal with several challenges caused by differences in the types and access patterns of data stored there. For example, one VM might use UniCache to store blocks from a slow tape drive, while another uses it to prefetch data from an SSD. The access patterns and size of objects stored in each of these caches may be similar, but clearly a cache miss that causes a read to the tape drive will have significantly higher cost. Alternatively, a third VM might use UniCache as a backend for memcached; here the cost of a cache miss may vary significantly depending on the database query that produced the data, as may the size.

III. UniCache Framework

UniCache is a generic volatile key-value store for both applications and OSes. We use DRAM memory as the first-layer cache and extend this to Flash memory for the second-level. In the current setup, we use a set of SSDs configured as a RAID array, and plan to evaluate other forms of Flash memory (e.g., PCI-based) in future work. In this section, we discuss the data store's interface and storage architecture.

Cache Interface: Our cache is structured as a key-value store, so we use the simple *get/put/invalidate* interface commonly used in systems such as memcached [19]. These functions are accessible via system calls (for user-space applications such as memcached) or hypercalls (for OS services like a disk cache). Applications that use these caches (e.g., web applications or any executable making disk accesses) do not need to be modified in any way. Figure 1 shows a sample deployment where both a disk caching system and memcached share access to UniCache. The Web App in VM-2 interacts using the standard memcached

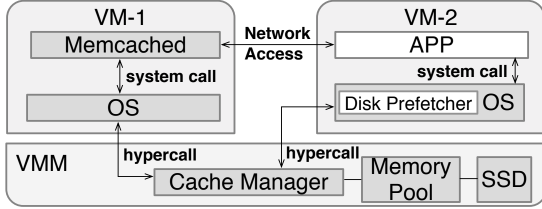


Figure 1. General UniCache deployment; UniCache interfaces with applications via a combination of a hypervisor call and a system call.

protocol with a modified memcached server running in VM-1. At the same time, the web app is running inside an OS that has a disk caching system that has been modified to store and retrieve data from UniCache. The Cache Manager will mediate access to the unified cache on behalf of these two VMs.

We have developed two applications to make use of this interface: a FUSE-based prefetching file system and a modified version of memcached. Unmodified applications can then either mount this file system or interact with memcached, and their data will be transparently stored within one of UniCache's storage layers.

Storage Architecture: Figure 2 illustrates UniCache's architecture. When a request is sent to the Cache Manager component, the data, if managed by UniCache, can either be directly accessed in a region of memory reserved by the hypervisor for the cache, or it may need to be retrieved from the SSDs. Since the Xen hypervisor does not contain device drivers, if the SSD storage layer must be accessed, the request must be passed to the UniCache Backend component running in the Domain-0 (Dom0, a special privileged domain). To communicate with the Dom0, the hypervisor must first create an event channel for notification, and shared memory and a descriptor ring for actual data transfer. Once there is an event in hypervisor that must put data into the SSDs, the Cache Manager notifies the Dom0 via an event channel, puts a request into the descriptor ring and data into shared memory. When the UniCache backend gets the notification via the event channel, it reads the descriptor ring and data, puts a response into the descriptor ring and data into shared memory, and notifies UniCache via a hypercall. A get request will complete in a reverse fashion.

IV. Cache Management Mechanisms

UniCache's Cache Manager component is responsible for deciding which objects to keep in each layer of the cache, which objects to evict when a layer is full, and how to partition the different layers among VMs.

A. Cache Replacement Algorithm

Many caches use locality schemes such as LRU to determine which objects to evict. However, LRU's effectiveness depends on the data stored in the cache being equivalent other than their access pattern. While this is true in a disk

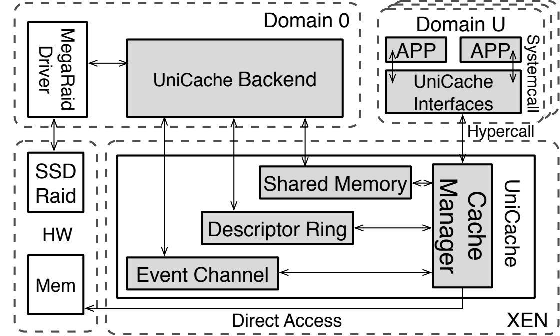


Figure 2. UniCache stores data either in hypervisor RAM or on an SSD RAID, which must be accessed via Dom0. Applications issue requests via a new system- / hyper-call interface.

cache, where all cached blocks are of identical size and will take a similar amount of time to read back from disk if they are evicted, it is not necessarily the case in more generic caches. Since UniCache's goal is to simultaneously cache data for a diverse set of use cases, we believe it is important to consider multiple factors in the cache replacement policy: locality, recovery time, and data size.

The benefits of using *temporal locality* to improve cache performance are well known, so here we focus on its limitations. Intuitively, every time a cache decides on an object to evict, it is making a cost-benefit trade off about how to use its resources. However, LRU only considers the potential *benefit* of keeping an object (the likelihood of it being accessed again), but does not consider either the cost of recovering the data if it needs to be brought into cache again or the relative cost of storing the object inside the cache. Nevertheless, temporal locality effectively captures the impact of application workloads, so UniCache assigns a locality score to each object j:

$$l_j = get_j/curr \in (0, 1] \tag{1}$$

where get_j is the last time the object was accessed, and $curr$ is the current time.

In addition to locality, UniCache also considers the *recovery cost* of each object. Most applications will initially check the cache for a piece of data, and if this fails, will read the data from its original source (e.g., the disk or database) before putting it into the cache. By measuring the time between the initial failed "get" operation and the subsequent "put", UniCache is able to estimate the amount of time it would take to recover a piece of data if it is ever evicted from the cache.

We have measured the caching behavior of three different applications: a social calendar web application that caches MySQL database query results and image files, a prefetching system that caches disk blocks for a video server, and a Wikipedia-based benchmark that caches both HTML content and database queries. Figure 3(a) illustrates the CDF of recovery times from each of our three sample applications.

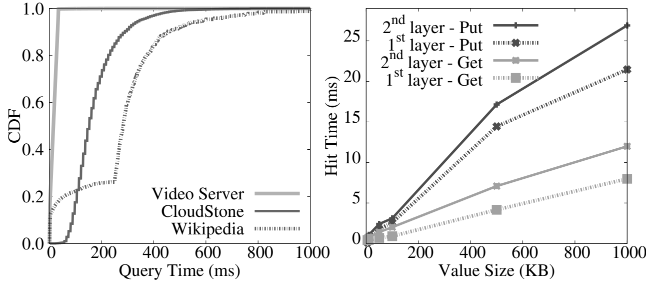


Figure 3. (a) The time to recover an object that has been evicted from a cache can vary widely. (b) The size of an object also impacts the overhead of storing it in layer 1 (DRAM) or layer 2 (SSD)

We find that the recovery time for both different applications and different objects within a single application can vary significantly. For example, nearly all of the video server blocks take approximately 5ms to recover since each is a simple disk read. In contrast, Wikipedia has a much wider range of recovery times, since much of the data stored in the cache is the result of multiple complex database queries. These results demonstrate that recovery time is an important metric that needs to be considered by caches storing heterogeneous data types. To account for this, UniCache assigns each object a recovery cost metric:

$$r_j = (put_j - check_j)/r_{max} \in (0, 1] \quad (2)$$

where r_{max} is the largest recovery cost among all the objects in the cache, $check_j$ is the time when an application first tried to retrieve data from the cache, and put_j is the time when an application pushed the data into the cache.

Finally, the *size of an object* impacts both the cost of storing it in the cache and the overhead of reading the data back out of each level of the cache. To demonstrate this, we have measured the time to put or get objects of different sizes into the memory and SSD-based layers of UniCache, as shown in Figure 3(b). The relative cost of accessing an object in RAM vs SSD actually decreases as the size rises, i.e., reading a 100KB object from SSD adds about 100% overhead compared to RAM, but this falls to 60% for a 1MB object. Of course, the SSD caching layer is also substantially larger than RAM. UniCache represents the size of each cached value as:

$$v_j = size_j/size_{max} \in (0, 1] \quad (3)$$

where $size_{max}$ is the maximum key-value length among objects in the cache.

The composite score of object j in VM i from Equations (1), (2), and (3) is defined as

$$score_{i,j} = \alpha_i \cdot l_{i,j} + \beta_i \cdot r_{i,j} + (1 - \alpha_i - \beta_i) \cdot v_{i,j} \quad (4)$$

where α_i and β_i, respectively, are locality sensitivity and recover time sensitivity parameters, and $\alpha_i + \beta_i \leq 1$. α_i and β_i are the knobs to control the importance depending on the type of a workload. We will show the impact of each parameter to each workload in our experiments. *When UniCache needs to replace an object from VM_i's cache, it finds the one with the lowest score based on Equation 4.* The selected object is moved from DRAM to SSD, or is dropped from the SSD if it has already been evicted once.

B. Cache Partitioning

The second key area explored by UniCache is how the hypervisor should partition its storage areas for multiple VMs.[1] These algorithms must be used both to divide up the memory region dedicated to the hypervisor cache and to partition the SSD resources. In effect, the cache partitioning algorithm determines which VM must run the cache eviction algorithm described above when there is insufficient space in either the DRAM or SSD cache, and, it helps prioritize VMs based on their workloads.

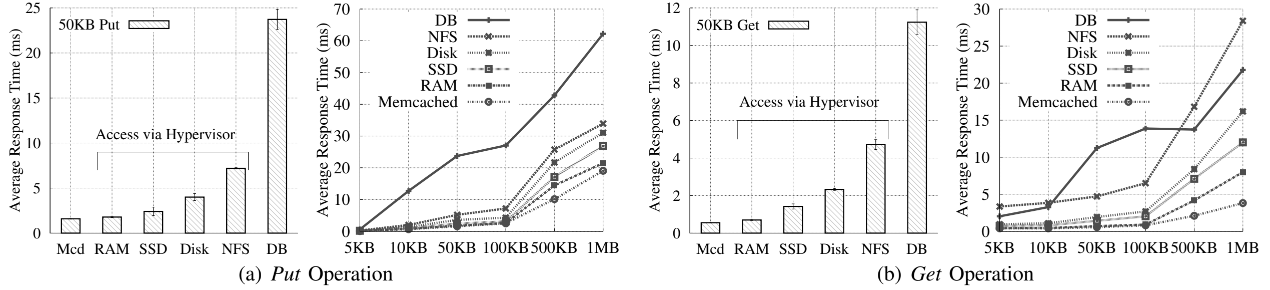
Best-Effort Cache Partitioning: In the simplest case, no explicit partitioning is performed, and UniCache's memory is offered to users on a "first come first served" basis, we call Best-Effort. This simple scheme does not account for either VM priority or performance.

Weight-Based Cache Partitioning: Our second cache partitioning scheme uses weights assigned to each VM to determine the relative portion of the cache they should have access to. 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.

Performance-Aware Cache Partitioning: While weights allow administrators to designate the priority of each VM, tuning the weights to provide *performance guarantees* can be very difficult. Our Performance-Aware cache partitioning scheme attempts to automate this process by adaptively adjusting the cache partitioning algorithm based on the performance metrics of each VM. UniCache is able to infer this information from measurements of hit rate and the recovery time of different request types—no additional application specific statistics or modifications are required. Together, these statistics allow UniCache to estimate the performance improvement that its cache is offering each application, so it can repartition its storage areas accordingly.

To adaptively adjust the cache size across multiple VMs, UniCache estimates the performance cost associated with reducing its cache size. We consider two factors that are the keys to optimize the system performance: the cache miss rate and the average recovery cost shown in Equation (2). The cache miss rate M_i of VM i ($M_i \in (0, 1)$) is the number of

[1] Our current implementation performs partitioning on a per-VM basis, but it would also be possible to subdivide a VM's allocated memory space for different cache applications by including an application identifier in all put/get requests.



(a) *Put* Operation (b) *Get* Operation

Figure 4. Overheads for basic operations of UniCache.

failed data fetches divided by the total number of requests. We also calculate the recovery cost running average for a stream of objects. That is, assuming there are already k objects and a new object $k+1$ comes in, the average recovery time R_i of VM i ($R_i \in (0,1)$) can be computed as:

$$R_{i,k+1} = (R_{i,k} \times k + r_{k+1})/(k+1) \qquad (5)$$

where r_{k+1} is the recovery time for object $k+1$ calculated using Equation (2). As a result, the final cost for VM i is:

$$C_i = M_i \times R_i \qquad (6)$$

Intuitively, the cost metric represents the performance benefit provided to each VM by the cache; a VM with a low cost must have either a low miss rate or only a minor performance impact when misses occur, and thus should experience a smaller overall performance degradation if its cache size is reduced.

Repartitioning must be performed when there is no sufficient cache space and the VMs are competing to get more room. UniCache seeks to equalize the performance estimates of all VMs[2]—when a new object is put in UniCache and there is no space to accommodate it, the Performance-Aware cache partitioning algorithm finds the VM with the minimum cost, i.e., the VM has low miss rate (high hit rate) and/or low recovery cost (fast source data retrieval), and likely will suffer a smaller performance degradation than other VMs. Subsequently, this VM is asked to surrender cache space for the new object.

Note that our Performance-Aware scheme could also account for differing VM priorities by applying a weight to each cost score.

V. Experimental Evaluation

Our goals for the evaluations are to see the overheads of UniCache through micro-benchmarks, and to check the performance for both UniCache-based memcached and prefetching through real workload-based benchmarks.

[2]Alternatively, weights could be easily added to the C_i metric to give varying performance benefits to different VMs based on importance.

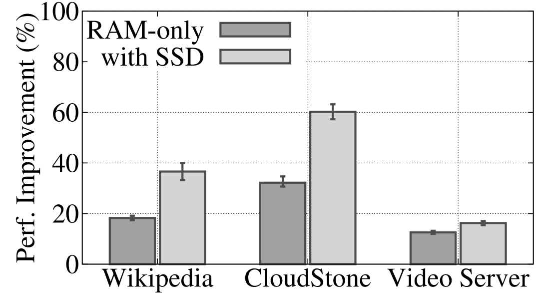


Figure 5. Performance Improvement with RAM and Flash

A. Environmental Setup

System Setup: Two experimental servers, each of which has 4× Intel Xeon X3450 2.67GHz processor, 16GB memory, a 500GB 7200RPM hard drive, and 4× 180GB Intel SSD 520 Series (SATA 6GB/s). Dom0 is deployed with Xen 4.1.2 and Linux kernel 3.5.0-23-generic, and the VMs use Linux kernel 3.3.1.

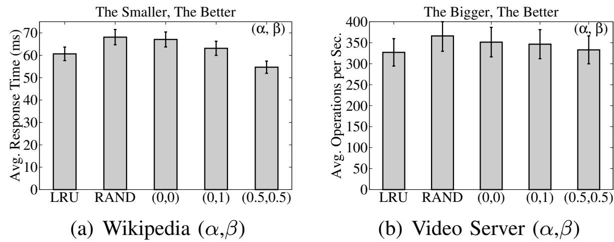
Benchmarks: We use realistic workloads to test the system: a video server [11] on a FUSE-based prefetching filesystem, Wikipedia with real request traces [25], and a social online calendar web app, CloudStone [23].

B. UniCache Overheads

Firstly, we identify the cost of accessing data with UniCache. Figure 4 shows the overheads of *put* and *get* operations across different storage areas including memcached (mcd) in user space, hypervisor-controlled RAM, SSD, Disk and NFS, or a database (DB). When the value size is 50KB, the *put* and *get* overheads of moving data between user space and hypervisor RAM instead of memcached are 0.1975 ms and 0.1445 ms, respectively; The *put* and *get* overheads between hypervisor RAM and SSD are 0.6266 ms and 0.7225 ms, respectively.

C. Cache Benefits

Figure 5 illustrates performance improvement over memcached for three applications when using 500MB memory cache and an SSD that is not space constrained. Wikipedia sees a 18% improvement with only memory, and an additional 18% with SSD support relative to the base case where there is no cache assistance. CloudStone shows a significant



(a) Wikipedia (α,β) (b) Video Server (α,β)

Figure 6. Comparison of Cache Replacement Algorithms for Wikipedia and Video Server with Prefetching; LRU = (1,0).

improvement – 32% with memory and an additional 28% with SSD support, since it stores both database queries and image files from file system. Video server with prefetching also shows performance improvement about 12% with memory only and extra 4% with SSD support. For these workloads, more than 4GB of data are populated on the SSDs. We expect that more flash memory will be utilized as the workloads increase.

D. Cache Replacement

We next test the impact of our cache replacement algorithms for both the prefetching video server and the Wikipedia workload. In the Wikipedia benchmark shown in Figure 6(a), we find that combining both LRU and recovery time information actually gives the best performance (lower response time is better) – an improvement of 10% over LRU and nearly 20% compared to the random eviction policy. This confirms our hypothesis that different application types can benefit from different caching policies, and that recovery time can be an important factor for applications that cache a mix of simple and complex query results.

The video server benchmark provides very different results. Figure 6(b) shows how the average operations per second change when using each algorithm. We find that the faster, random eviction algorithm actually gives the best performance. However, this is not surprising since the size and recovery time metrics of every object in the cache is nearly identical since they are all disk blocks. Similarly, since the videoserver mostly performs sequential reads, LRU is not a good eviction policy. As a result, none of the replacement algorithms are statistically better due to high variance in throughput.

E. Cache Partitioning

In this section we study UniCache's partitioning schemes by using a small 100MB first-level cache for three competing VMs. Wikipedia starts to fill up the cache, followed by CloudStone at some point later, and a video server with prefetching capability last. To highlight the effects of various partitioning schemes, in this test UniCache uses the LRU replacement algorithm.

In Figure 7, the first row of the figure demonstrates cache size changes over time (600 seconds), and the second row shows miss rates of each VM. As shown in Figure 7(a), the Video Server application is very aggressive, so it dominates the cache when using the Best-Effort algorithm, so that Wikipedia and CloudStone do not get enough cache space. The Weight-Based partitioning algorithm (Figure 7(b)) can equalize the portion given to each VM, causing CloudStone and Video Server to maintain a similar size, and Wikipedia to keep increasing until it obtains the equal amount as others. However, an equal weight does not result in an equal miss-rate, as shown in 7(e), and this in turn means that performance measured in response time will vary (Figure 8).

The Best-Effort scheme only benefits the Video Server, causing significant performance issues for both Wikipedia and CloudStone (Figure 8 (a) and (b)). The Weight based scheme is acceptable for Wikipedia, since its size is smaller than its weight. However, CloudStone suffers from the Weight-Based scheme because it cannot store some important data in the first level of the cache. The Performance-Aware scheme provides the best performance for both Wikipedia and CloudStone, but the Video Server benchmark cannot reach its peak performance. We believe that Performance-Aware is providing a good trade-off for each application: it gives CloudStone more space than the Video Server because misses for the CloudStone application have greater cost. In this experiment, Wikipedia gradually increases its share, and we expect that it will continue to take space away from the video server (and possibly CloudStone), since it has some objects with very high recovery cost, as was shown in Figure 3(a).

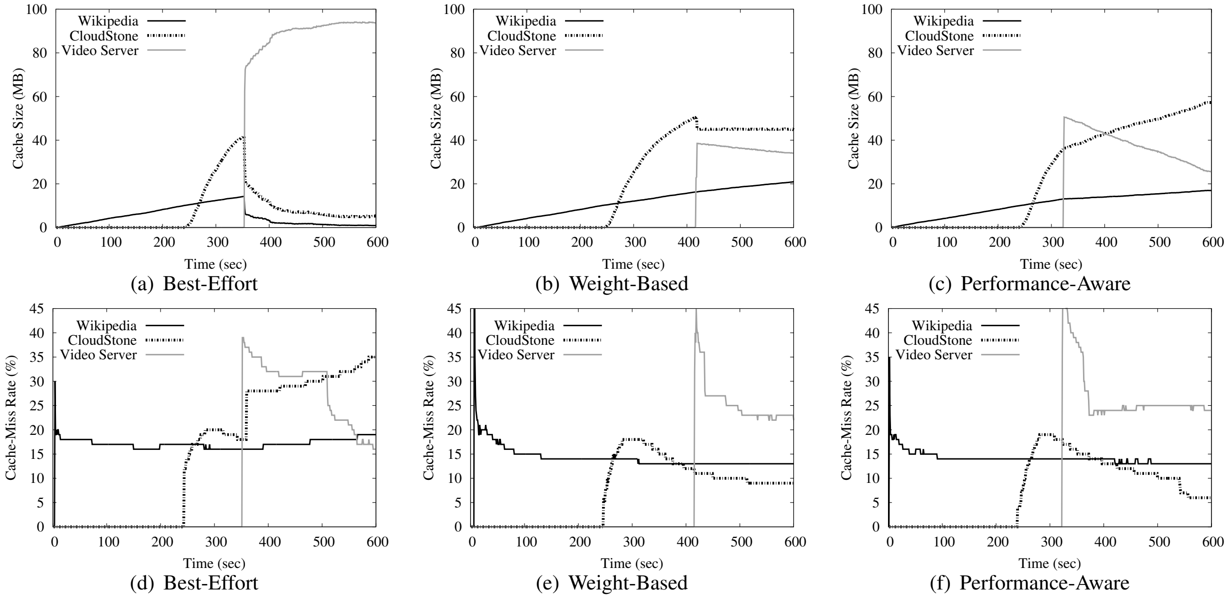
Table I
RESPONSE TIMES FOR THREE PARTITIONING ALGORITHMS AND THREE APPLICATIONS

Response Time	Best-Effort	Weight	Perf-Aware
Wikipedia	84 ms	36 ms	36 ms
CloudStone	12 sec	400 ms	170 ms
VideoServer	4 ms	7 ms	7 ms

Table I summarizes the average response time over the last 100 seconds of the experiment, when the partitions have somewhat stabilized. Wikipedia shows a 234% better performance with the Performance-Aware partitioning algorithm and Weight-Based algorithm compared to the Best-Effort algorithm. CloudStone experiences performance anomalies in the Best-Effort case since the cache performs so poorly, but we see that the Performance-Aware approach provides a significant benefit compared to the Weight-Based algorithm. Only VideoServer shows better performance under Best-Effort, but clearly that comes at a high cost to the other applications.

VI. RELATED WORK

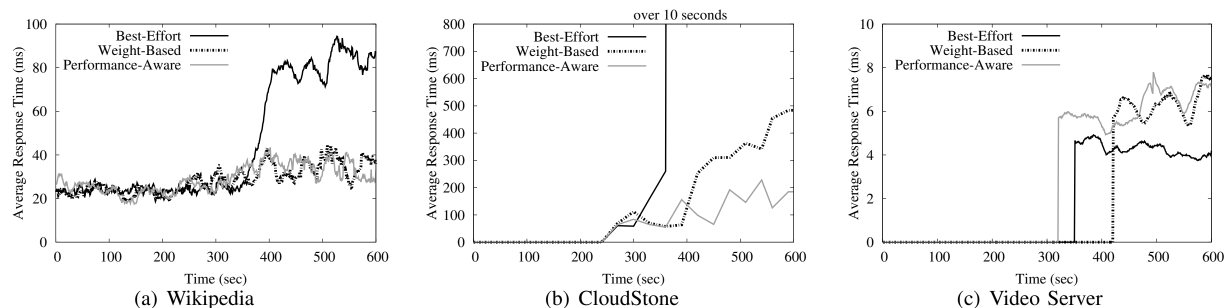
UniCache proposes a unified cache server at the hypervisor level to offer more flexible cache allocation to operating



(a) Best-Effort (b) Weight-Based (c) Performance-Aware

(d) Best-Effort (e) Weight-Based (f) Performance-Aware

Figure 7. Cache size vs. cache-miss rate for three partitioning algorithms (Best-Effort, Weight-Based, Performance-Aware) and three applications (Wikipedia, CloudStone, Video Server).



(a) Wikipedia (b) CloudStone (c) Video Server

Figure 8. Average response time with three different partitioning algorithms shows the performance impact of three applications: Wikipedia, CloudStone, and Video server.

systems and applications. This work builds on top of our Mortar infrastructure [13], which focused on aggregating spare memory among a group of VMs; we extend that here with support for storing data in DRAM and SSDs, and by providing new cache eviction and repartitioning algorithms.

These challenges have been tackled indirectly through dynamic virtual machine memory management and hypervisor or SSD based cache systems. Waldspurger [26] seeks to manipulate the allocation of each VM's memory based on its needs, but this cannot differentiate between memory pages used to store critical data and those used as a cache. UniCache enables the hypervisor to differentiate volatile data from other types, and also extends the storage area that can be used to include SSDs. Zhu et al. [27] dynamically adjust the number of cache servers based on statistical inference, but only focus on one type of caching application. Levandoski et al. [15] propose classification algorithms to identify hot and cold data in main memory databases, but the process includes an offline analysis so that it is hard to apply for the environments with varying workloads.

Several prior projects have focused on using SSDs or hypervisor memory to improve the performance of a particular cache, e.g., a VM's disk cache [17] or a database's query cache [5], [9], [18]. A key aspect of UniCache is that it can be used to cache data from a wide variety of sources, and thus it needs to use more information about its data objects to determine where to store them (memory or SSD) and how to guide eviction policies. We also focus on multi-tenant environments that must specifically deal with partitioning the cache among users.

VII. Conclusion

Managing memory in a virtualized environment is difficult since the hypervisor does not know how memory is being used within each VM. We have developed UniCache to transfer the management of volatile data directly to the hypervisor. UniCache stores data in either main memory or on an SSD RAID array. Since UniCache is designed to store data for a wide variety of applications simultaneously (e.g., disk blocks and web database queries), we have developed an advanced cache replacement policy that can account for both object popularity and its cost to bring back into the cache if it is needed later. We believe that UniCache holds

promise as a way to more effectively share multiple layers of the storage hierarchy among competing VMs.

Acknowledgments

We thank the reviewers for their help improving this paper. This work was supported in part by NSF grants CNS-1253575, CNS-1350766, OCI-0937875, National Natural Science Foundation of China under Grant No. 61370059 and No. 61232009, and Beijing Natural Science Foundation under Grant No. 4122042.

References

[1] Windows Azure. http://www.windowsazure.com/en-us/services/cache.

[2] Anirudh Badam and Vivek S. Pai. Ssdalloc: hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 16–16, Berkeley, CA, USA, 2011. USENIX Association.

[3] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. Hashcache: cache storage for the next billion. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 123–136, Berkeley, CA, USA, 2009. USENIX Association.

[4] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 10–22, New York, NY, USA, 1992. ACM.

[5] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. Ssd bufferpool extensions for database systems. *Proc. VLDB Endow.*, 3(1-2):1435–1446, September 2010.

[6] Amazon Corporation. http://aws.amazon.com/elasticache.

[7] Biplob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 16–16, Berkeley, CA, USA, 2010. USENIX Association.

[8] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. Ulcc: a user-level facility for optimizing shared cache performance on multicores. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 103–112, New York, NY, USA, 2011. ACM.

[9] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. Turbocharging dbms buffer pool using ssds. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1113–1124, New York, NY, USA, 2011. ACM.

[10] K. Eshraghian, Kyoung-Rok Cho, O. Kavehei, Soon-Ku Kang, D. Abbott, and Sung-Mo Steve Kang. Memristor mos content addressable memory (mcam): Hybrid architecture for future high performance search engines. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(8):1407–1417, Aug 2011.

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

[12] Xiaoming Gu and Chen Ding. On the theory and potential of lru-mru collaborative cache management. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 43–54, New York, NY, USA, 2011. ACM.

[13] Jinho Hwang, Ahsen Uppal, Timothy Wood, and Howie Huang. Mortar: Filling the gaps in data center memory. *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2014.

[14] Jinho Hwang and Timothy Wood. Adaptive performance-aware distributed memory caching. *International Conference on Autonomic Computing*, 2013.

[15] Justin J. Levandoski, Per-Ake Larson, and Radu Stoica. Identifying hot and cold data in main-memory databases. *International Conference on Data Engineering (ICDE)*, 2013.

[16] Ke Liu, Xuechen Zhang, Kei Davis, and Song Jiang. Synergistic coupling of ssd and hard disk for qos-aware virtual memory. *IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

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

[18] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proc. VLDB Endow.*, 5(10):1076–1087, June 2012.

[19] Memcached. http://memcached.org.

[20] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[21] Xiangyong Ouyang, Nusrat S. Islam, Raghunath Rajachandrasekar, Jithin Jose, Miao Luo, Hao Wang, and Dhabaleswar K. Panda. Ssd-assisted hybrid memory to accelerate memcached over high performance networks. In *Proceedings of the 2012 41st International Conference on Parallel Processing*, ICPP '12, pages 470–479, Washington, DC, USA, 2012. IEEE Computer Society.

[22] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 325–334, New York, NY, USA, 2013. ACM.

[23] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, O Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0, 2008.

[24] Christopher Stewart, Aniket Chakrabarti, and Rean Grifth. Zoolander: Efciently meeting very strict, low-latency slos. *International Conference on Autonomic Computing*, 2013.

[25] Erik-Jan van Baaren. Wikibench: A distributed, wikipedia based web application benchmark. *Master Thesis*, 2009.

[26] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[27] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Saving cash by using less cache. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.