Virtual Machine Memory Access Tracing With Hypervisor Exclusive Cache*

Pin Lu and Kai Shen
Department of Computer Science, University of Rochester
{pinlu, kshen}@cs.rochester.edu

Abstract

Virtual machine (VM) memory allocation and VM consolidation can benefit from the prediction of VM page miss rate at each candidate memory size. Such prediction is challenging for the hypervisor (or VM monitor) due to a lack of knowledge on VM memory access pattern. This paper explores the approach that the hypervisor takes over the management for part of the VM memory and thus all accesses that miss the remaining VM memory can be transparently traced by the hypervisor.

For online memory access tracing, its overhead should be small compared to the case that all allocated memory is directly managed by the VM. To save memory space, the hypervisor manages its memory portion as an exclusive cache (*i.e.*, containing only data that is not in the remaining VM memory). To minimize I/O overhead, evicted data from a VM enters its cache directly from VM memory (as opposed to entering from the secondary storage). We guarantee the cache correctness by only caching memory pages whose current contents provably match those of corresponding storage locations. Based on our design, we show that when the VM evicts pages in the LRU order, the employment of the hypervisor cache does not introduce any additional I/O overhead in the system.

We implemented the proposed scheme on the Xen para-virtualization platform. Our experiments with microbenchmarks and four real data-intensive services (SPECweb99, index searching, TPC-C, and TPC-H) illustrate the overhead of our hypervisor cache and the accuracy of cache-driven VM page miss rate prediction. We also present the results on adaptive VM memory allocation with performance assurance.

1 Introduction

Virtual machine (VM) [2,8,22] is an increasingly popular service hosting platform due to its support for fault containment, performance isolation, ease of transparent system management [4] and migration [7]. For data-intensive services, the problem of VM memory allocation arises in the context of multi-VM memory sharing and service consolidation. To achieve performance isolation with quality-of-service (QoS) constraints, it is desirable to predict the VM performance (or page miss rate) at each candidate memory size. This information is also called page miss ratio curve [27].

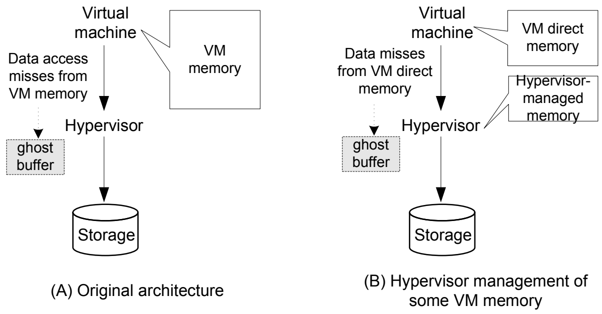

Figure 1: Virtual machine architecture on which the hypervisor manages part of the VM memory. Less VM direct memory results in more hypervisor-traceable data accesses.

Typically, the hypervisor (or the VM monitor) sees all VM data accesses that miss the VM memory in the form of I/O requests. Using the ghost buffer [12, 17] technique, it can predict the VM page miss rate for memory sizes beyond its current allocation. However, since data accesses that hit the VM memory are not visible to the hypervisor, it is challenging to estimate VM page miss rate for memory sizes smaller than its current allocation. An intuitive idea to predict more complete VM page miss rate information is the following (illustrated in Figure 1). The hypervisor takes over the management for part of the VM memory and thus all accesses that miss the remaining VM directly-managed memory (or *VM direct memory*) can be transparently traced by the hypervisor. By applying the same ghost buffer technique, the hypervisor can now predict VM performance for memory sizes beyond the VM direct memory size.

To be able to apply online, our VM memory access tracing technique must be efficient. More specifically, the hypervisor memory management should deliver competitive performance compared to the original case that all allocated VM memory is directly managed by the VM OS. In order to avoid double caching, we keep the hypervisor memory as an exclusive cache to the VM direct

*This work was supported in part by the National Science Foundation (NSF) grants CCR-0306473, ITR/IIS-0312925, CNS-0615045, CCF-0621472, NSF CAREER Award CCF-0448413, and an IBM Faculty Award.

memory. Exclusive cache [5,12,25] typically admits data that is just evicted from its upper-level cache in the storage hierarchy (VM direct memory in our case). It is efficient for evicted VM data to enter directly into the hypervisor cache (as opposed to loading from secondary storage). However, this may introduce caching errors when the VM memory content does not match that of corresponding storage location. We ensure the correctness of our cache by only admitting data that has provably the same content as in the corresponding storage location. We achieve this by only accepting evicted pages before reuse to avoid data corruptions, and by maintaining two-way mappings between VM memory pages and storage locations to detect mapping changes in either direction.

Based on our design, our hypervisor exclusive cache is able to manage large chunk of a VM's memory without increasing the overall system page faults (or I/O overhead). However, the cache management and minor page faults (*i.e.*, data access misses at the VM direct memory that subsequently hit the hypervisor cache) incur some CPU overhead. We believe that the benefit of predicting accurate VM page miss ratio curve outweighs such overhead in many situations, particularly for data-intensive services where I/O is a more critical resource than CPU. Additionally, when the hypervisor cache is employed for acquiring VM memory access pattern and guiding VM memory allocation, it only needs to be enabled intermittently (when a new memory allocation is desired).

2 Related Work

VM memory allocation Virtual machine (VM) technologies like Disco [3, 9], VMware [8, 22, 23], and Xen [2] support fault containment and performance isolation by partitioning physical memory among multiple VMs. It is inherently challenging to derive good memory allocation policy at the hypervisor due to the lack of knowledge on VM data access pattern that is typically available to the OS. Cellular Disco [9] supports memory borrowing from cells rich of free memory to memory-constrained cells. However, cells in their context are VM containers and they are more akin to physical machines in a cluster. Their work does not address policy issues for memory allocation among multiple VMs within a cell.

In VMware ESX server, Waldspurger proposed a sampling scheme to transparently learn the proportion of VM memory pages that are accessed within a time period [23]. This result can be used to derive a working set estimation and subsequently to guide VM memory allocation. This sampling approach requires very little overhead but it is less powerful than VM memory allocation based on accurate VM page miss ratio curve.

1. The sampling approach may not be able to support memory allocation with flexible QoS constraint. One possible allocation objective is to minimize a system-wide page miss rate metric with the constraint that no VM may have more than $\delta\%$ increase in page misses compared to its baseline allocation.
2. Although the sampling approach can estimate the amount of memory accessed within a period of time (*i.e.*, the working set), the working set may not always directly relate to the VM's performance behavior. For example, it is known that the working set model may over-estimate the memory need of a program with long sequential scans [6, 18].

We will elaborate on these issues in Section 4.2 and experimentally demonstrate them in Section 6.5.

Exclusive cache management Wong and Wilkes [25] argued that exclusive lower-level caches are more effective than inclusive ones (by avoiding double caching). This is particularly the case when lower-level caches are not much larger than upper-level ones in the storage hierarchy. To implement an exclusive cache, they introduced a DEMOTE operation to notify lower-level caches about data evictions from upper-level caches. To achieve cache correctness, they assume the evicted data contains exactly the same content as in the corresponding storage location. They do not address how this is achieved in practice.

Chen *et al.* [5] followed up Wong and Wilkes's work by proposing a transparent way to infer upper-level cache (memory page) evictions. By intercepting all I/O reads/writes, they maintain the mapping from memory pages to storage blocks. A mapping change would indicate a page reuse which infers an eviction has occurred earlier. Jones *et al.* [12] further strengthened the transparent inference of page reuses by considering additional issues such as storage block liveness, file system journaling, and unified caches (virtual memory cache and file system buffer cache). In these designs, the exclusive cache is assumed to be architecturally closer to the lower-level storage devices and data always enters the cache from the storage devices. In our context, however, it is much more efficient for evicted data to enter the hypervisor cache directly from VM memory. This introduces potential correctness problems when the entering VM memory content does not match the storage content. In particular, both Chen *et al.* [5] and Jones *et al.* [12] detect page reuses and then infer earlier evictions. At page reuse time, the correct content for the previous use may have already been zero-cleaned or over-written — too late for loading into the hypervisor cache.

Hypervisor-level cache As far as we know, existing hypervisor-level buffer cache (*e.g*, Copy-On-Write disks in Disco [3] and XenFS [24]) is used primarily for the purpose of keeping single copy of data shared across multiple VMs. At the absence of such sharing, one common belief is that buffer cache management is better left to the VM OS since it knows more about the VM itself. In this paper, we show that the hypervisor has sufficient information to manage the cache efficiently since all accesses to the cache are trapped in software. More importantly, the cache provides a transparent means to learn the VM data access pattern which in turn guides performance-assured memory allocation.

OS-level program memory need estimation Zhou *et al.* [27] and Yang *et al.* [26] presented operating system (OS) techniques to estimate program memory need for achieving certain desired performance. The main idea of their techniques is to revoke access privilege on (infrequently accessed) partial program memory and trap all accesses on this partial memory. Trapped data accesses are then used to estimate program page miss ratio or other performance metric at different memory sizes. While directly applying these OS-level technique within each VM can estimate VM memory need, our hypervisor-level approach attains certain advantages while it also presents unique challenges.

- *Advantages*: In a VM platform, due to potential lack of trust between the hypervisor and VMs, it is more appropriate for the hypervisor to collect VM memory requirement information rather than let the VM directly report such information. Further, given the complexity of OS memory management, separating the memory need estimation from the OS improves the whole system modularity.
- *Challenges*: Correctly maintaining the hypervisor exclusive cache is challenging due to the lack of inside-VM information (*e.g.*, the mapping information between memory pages and corresponding storage locations). Such information is readily available for an OS-level technique within each VM. Further, the employment of a hypervisor buffer cache potentially incurs more management overhead. More careful design and implementation are needed to keep such overhead small.

3 Hypervisor-level Exclusive Cache

Our hypervisor-level cache has several properties that are uncommon to general buffer caches in storage hierarchies. First, its content is exclusive to its immediate upper-level cache in the storage hierarchy (VM direct memory). Second, this cache competes for the same physical space with its immediate upper-level cache. Third, data enters this cache directly from its immediate upper-level cache (as opposed to entering from its immediate lower-level cache in conventional storage systems). These properties combined together present unique challenges for our cache design. In this section, we present the basic design of our hypervisor-level exclusive cache, propose additional support to ensure cache correctness, discuss the transparency of our approach to the VM OS, and analyze its performance (cache hit rate) and management overhead.

3.1 Basic Design

The primary design goals of our hypervisor cache are that: 1) it should try not to contain any data that is already in VM memory (or to be exclusive); and 2) it should try to cache data that is most likely accessed in the near future. To infer the access likelihood of a page, we can use the page eviction order from the VM as a hint. This is because the VM OS would only evict a page when the page is believed to be least useful in the near future.

For write accesses, we can either support delayed writes (*i.e.*, writes are buffered until the buffered copies have to be evicted) or write-through in our cache management. Delayed writes reduce the I/O traffic, however delayed writes in the hypervisor cache are not persistent over system crashes. They may introduce errors over system crashes when the VM OS counts the write completion as a guarantee of data persistence (*e.g.*, in file system `fsync()`). Similar problems were discussed for delayed writes in disk controller cache [16]. On the other hand, the SCSI interface allows the OS to individually specify I/O requests with persistence requirement (through the force-unit-access or FUA bit). In general, we believe delayed writes should be employed whenever possible to improve performance. However, write-through might have to be used if persistence-related errors are not tolerable and we cannot distinguish those writes with persistence requirement and those without.

In our cache management, all data units in the hypervisor cache (typically memory pages) are organized into a queue. Below we describe our management policy, which defines actions when a read I/O request, a VM data eviction, or a write I/O request (under write-through or delayed write) reaches the hypervisor. A simplified illustration is provided in Figure 2.

- Read I/O request $\Longrightarrow$ If it hits the hypervisor cache, then we bring the data from the hypervisor cache to VM memory and return the I/O request. To avoid double caching at both levels, we move it to the queue head (closest to being discarded) in the hypervisor cache or explicitly discard it. If the request misses at the hypervisor cache, then we bring the data from external storage to the VM memory as usual. We do *not* keep a copy in the hypervisor cache.
- VM data eviction $\Longrightarrow$ We cache the evicted data at the queue tail (furthest away from being discarded)

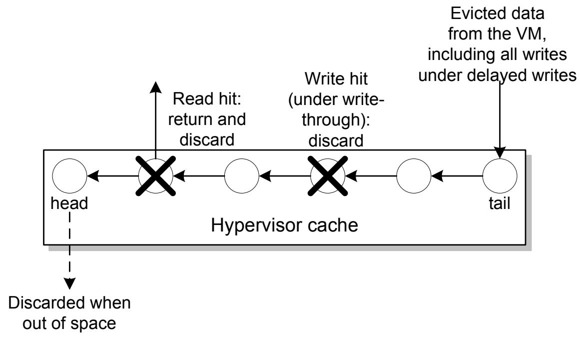

Figure 2: A simplified illustration of our hypervisor cache management.

of the hypervisor cache. In this way, the hypervisor cache discards data in the same order that the VM OS evicts them. If the hypervisor cache and the VM memory are not strictly exclusive, it is also possible for the evicted data to hit a cached copy. In this case, we simply move the cached copy to the queue tail.

- Write I/O request (write-through) $\Longrightarrow$ We write the data to the external storage as usual. If the request hits the hypervisor cache, then we also discard the hypervisor cache copy. We do not need to keep an updated copy since the VM memory should already contain it.
- Write I/O request (delayed write) $\Longrightarrow$ Each write I/O request is buffered at the hypervisor cache (marked as dirty) and then the request returns. The data is added at the hypervisor cache queue tail. If the request hits an earlier cached unit on the same storage location, we discard that unit. Although the write caching creates temporary double buffering in VM memory and hypervisor cache, this double buffering is of very short duration if the VM OS also employs delayed writes (in this case a write is typically soon followed by an eviction). Dirty cached data will eventually be written to the storage when they reach the queue head to be discarded.

To support lookup, cached entries in the hypervisor cache are indexed according to their mapped storage locations. Therefore we need to know the mapped storage location for each piece of evicted data that enters the cache. Such mapping can be constructed at the hypervisor by monitoring I/O requests between VM memory pages and storage locations [5]. Specifically, an I/O (read or write) between page p and storage location s establishes a mapping between them (called *P2S* mapping). A new mapping for a page replaces its old mapping. Additionally, we delete a mapping when we are aware that it becomes stale (*e.g.*, when a page is evicted or released). We distinguish page release from page eviction in that page eviction occurs when the VM runs out of memory space while page release occurs when the VM OS feels it is not useful. For example, a page in file system buffer cache is released when its mapped storage location becomes invalid (*e.g.*, as a result of file truncation). Although both page eviction and release should delete the page to storage location mapping, only evicted data should enter the hypervisor cache. Figure 3 provides an illustration of the VM page to storage location mapping table.

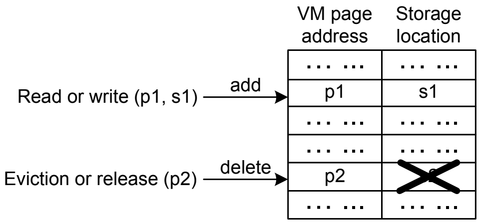

Figure 3: An illustration of the VM page to storage location mapping table (or P2S table in short).

3.2 Cache Correctness

Since the hypervisor cache directly supplies data to a read request that hits the cache, the data must be exactly the same as in the corresponding storage location to guarantee correctness. To better illustrate our problem, below we describe two realistic error cases that we experienced:

Missed eviction/release: Page p is mapped to storage location s and the hypervisor is aware of this mapping. Later, p is reused for some other purpose but the hypervisor fails to detect the page eviction or release. Also assume this reuse slips through the detection of available reuse detection techniques (*e.g.*, Geiger [12]). When p is evicted again and the hypervisor captures the eviction this time, we would incorrectly admit the data into the cache with mapped storage location s. Later read of s will hit the cache and return erroneous data.

Stale page mapping: The VM OS may sometimes keep a page whose mapping to its previously mapped storage location is invalid. For instance, we observe that in the Linux 2.6 ext3 file system, when a meta-data block is recycled (due to file deletion for example), its memory cached page would remain (though inaccessible from the system). Since it is inaccessible (as if it is a leaked memory), its consistency does not need to be maintained. At its eviction time, its content may be inconsistent with the storage location that it was previously mapped to. Now if we cache them, we may introduce incorrect data into the hypervisor cache. The difficulty here is that without internal OS knowledge, it is hard to tell whether an evicted page contains a stale page mapping or not.

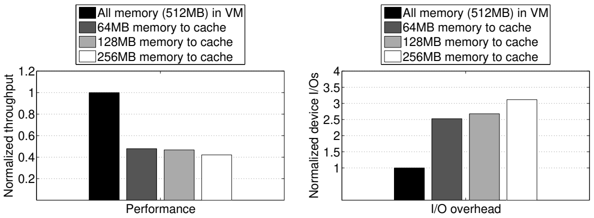

Figure 4: Service throughput reduction and real device I/O increase of the hypervisor cache management when new cached data is always loaded from the storage. Results are for a trace-driven index searching workload (described in Section 6.1). Due to such large cost, our cache management allows evicted VM data to enter the cache directly from VM memory.

We identify two sufficient conditions that would meet our cache correctness goal:

Condition 1 (admission): Each time we cache a page, we guarantee it contains the same content as in the corresponding storage location.

Condition 2 (invalidation): The hypervisor captures all I/O writes that may change storage content. If we have a cached copy for the storage location being written to, we remove it from the hypervisor cache.

The invalidation condition is easy to support since the hypervisor typically captures all I/O requests from VMs to storage devices. The challenge lies in the support for the admission condition due to the limited amount of information available at the hypervisor. Note that the admission condition is trivially satisfied if we always load new data from the storage [5,12]. However, this is inappropriate in our case due to its large cost. To illustrate the cost when data enters the cache from the storage, Figure 4 shows the substantial performance reduction and I/O increase for a trace-driven index searching workload.

Our correctness guarantee is based on a set of assumptions about the system architecture and the VM OS. First, we assume that the hypervisor can capture all read/write I/O operations to the secondary storage. Second, we assume the hypervisor can capture every page eviction and release (before reuse) in the VM. Third, we assume no page is evicted from the VM while it is dirty (*i.e.*, it has been changed since last time it was written to the storage). While the first and the third assumptions are generally true without changing the VM OS, the second assumption needs more discussion and we will provide that later in Section 3.3.

In addition to these assumptions, we introduce a reverse mapping table that records the mapping from storage locations to VM pages (or S2P table). Like the P2S mapping table described in Section 3.1, a mapping is established in the S2P table each time a read/write I/O request is observed at the hypervisor. A new mapping for a storage location replaces its old mapping. Each time a page p is evicted from the VM memory, we check the two mapping tables. Let p be currently mapped to storage location s in the P2S table and s be mapped to page p' in the S2P table. We admit the evicted page into the hypervisor cache only if $p = p'$. This ensures that p is the last page that has performed I/O operation on storage location s.

Correctness proof: We prove that our approach can ensure the admission condition for correctness. Consider each page p that we admit into the hypervisor cache with a mapped storage location s. Since we check the two mapping tables before admitting it, the most recent I/O (read or write) that concerns p must be on s and the reverse is also true. This means that the most recent I/O operation about p and the most recent I/O operation about s must be the same one. At the completion of that operation (no matter whether it is a read or write), p and s should contain the same content. Below we show that neither the storage content nor the page content has changed since then. The storage content has not changed since it has not established mapping with any other page (otherwise the S2P table would have shown it is mapped to that page). The page content has not changed because it has not been reused and it is not dirty. It is not reused since otherwise we should have seen its eviction or release before reuse and its mapping in the P2S table would have been deleted. Note our assumption that we can capture every page eviction and release in the VM. ■

3.3 Virtual Machine Transparency

It is desirable for the hypervisor cache to be implemented with little or no change to the VM OS. Most of our design assumptions are readily satisfied by existing OSes without change. The only non-transparent aspect of our design is that the hypervisor must capture every page eviction and release (before reuse) in the VM. A possible change to the VM OS is to make an explicit trap to the hypervisor at each such occasion. The only information that the trap needs to provide is the address of the page to be evicted or released.

The suggested change to the VM OS is semantically simple and it should be fairly easy to make for existing OSes. Additionally, the eviction or release notification should not introduce additional fault propagation vulnerability across VM boundaries. This is because the only way this operation can affect other VMs' correctness is when multiple VMs are allowed to access the same storage locations. In this case a VM can always explicitly write invalid content into these shared locations. In summary, our suggested change to the VM OS fits well into a para-virtualization platform such as Xen [2].

We also provide some discussions on the difficulty of implementing a hypervisor-level exclusive cache in a fully transparent way. Earlier transparent techniques [5, 12] can detect the eviction of a page after its reuse. However, reuse time detection is too late for loading evicted data directly from VM memory to the hypervisor cache. At reuse time, the original page content may have already be changed and some OSes would have zeroed the page before its reuse. Further, it is not clear any available transparent technique can capture every page reuse without fail (no false negative).

3.4 Performance and Management Overhead

The primary goal of the hypervisor cache is to allow transparent data access tracing. Yet, since it competes for the same physical space with the VM direct memory, its employment in an online system should not result in significant VM performance loss. This section analyzes the cache hit rate and management overhead of our hypervisor cache scheme compared to the original case in which all memory is directly managed by the VM.

Caching performance We compare the overall system cache hit rate of two schemes: the first contains a VM memory of X pages with an associated hypervisor cache of Y pages (called *Hcache* scheme); the other has a VM memory of $X+Y$ pages with no hypervisor cache (called *VMonly* scheme). Here we consider an access to be a hit as long as it does not result in any real device I/O. We use an ideal model in which the VM OS employs perfect LRU cache replacement policy. Under this model, we show that Hcache and VMonly schemes achieve the same cache hit rate on any given data access workload. Our result applies to both read and write accesses if we employ delayed writes at the hypervisor cache. Otherwise (if we employ write-through) the result only applies to reads.

Consider a virtual LRU stack [15] that orders all pages according to their access recency — a page is in the k-th location from the top of the stack if it is the k-th most recently accessed page. At each step of data access, the VM memory under the VMonly scheme contains the top $X + Y$ pages in the virtual LRU stack. For the Hcache scheme, the top X pages in the stack are in the VM memory while the next Y pages should be in the hypervisor cache. This is because our hypervisor cache is exclusive to the VM memory and it contains the most recently evicted pages from the VM memory (according to the cache management described in Section 3.1). So the aggregate in-memory content is the same for the two schemes at each step of data access (as shown in Figure 5). Therefore VMonly and Hcache should have the identical data access hit/miss pattern for any given workload and consequently they should achieve the same cache hit rate.

The above derivation assumes that the hypervisor

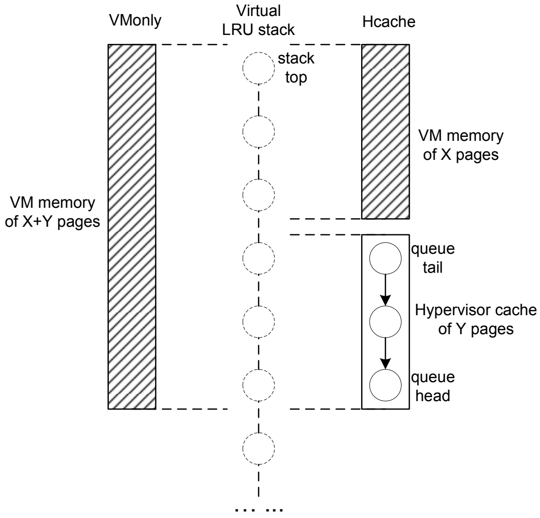

Figure 5: In-memory content for VMonly and Hcache when the VM OS employs perfect LRU replacement.

cache is strictly exclusive to the VM memory and all evicted pages enter the hypervisor cache. These assumptions may not be true in practice for the following reasons. First, there is a lag time between a page being added to the hypervisor and it is being reused in the VM. The page is doubly cached during this period. Second, we may prevent some evicted pages from entering the hypervisor cache due to correctness concern. However, these exceptions are rare in practice so that they do not visibly affect the cache hit rate (as demonstrated by our experimental results in Section 6.3).

Management overhead The employment of the hypervisor cache introduces additional management (CPU) overhead, including the mapping table lookup and simple queue management in the hypervisor cache. Additionally, the handling of page eviction and minor page fault (*i.e.*, data access misses at the VM memory that subsequently hit the hypervisor cache) requires data transfer between the VM memory and the hypervisor cache. Page copying can contribute a significant amount of overhead. Remapping of pages between the VM and the hypervisor cache may achieve the goal of data transfer with much less overhead. Note that for minor page faults, there may also be additional CPU overhead within the VM in terms of page fault handling and I/O processing (since this access may simply be a hit in VM memory if all memory is allocated to the VM).

For the employment of the hypervisor cache, the benefit of transparent data access is attained at the additional CPU cost. More specifically, when we move away Y pages from the VM memory to the hypervisor cache, we can transparently monitor data accesses on these pages while at the same time we may incur overhead of cache management and minor page faults on them.

Note that the monitoring of these Y pages provides more useful information than monitoring Y randomly chosen pages [23]. This is because the pages in the hypervisor cache are those that the VM OS would evict first when its memory allocation is reduced. Access statistics on these pages provide accurate information on additional page misses when some memory is actually taken away.

4 Virtual Machine Memory Allocation

With the hypervisor management for part of the VM memory, we discuss our ability to predict more complete VM page miss ratio curve and consequently to guide VM memory allocation. We then provide an example of complete-system workflow for our guided VM memory allocation. We also discuss a potential vulnerability of our adaptive memory allocation to VM manipulation.

4.1 VM Miss Ratio Curve Prediction

To best partition the limited memory for virtual machines (VMs) on a host or to facilitate VM consolidation over a cluster of hosts, it is desirable to know each VM's performance or page miss rate at each candidate allocation size (called miss ratio curve [27]). Jones *et al.* [12] showed that the miss ratio curve can be determined for memory sizes larger than the current memory allocation when all I/O operations and data evictions of the VM are traced or inferred. Specifically, the hypervisor maintains a ghost buffer (a simulated buffer with index data structure but no actual page content) [17]. Ghost buffer entries are maintained in the LRU order and hypothetical hit counts on each entry are tracked. Such hit statistics can then be used to estimate the VM page hit rate when the memory size increases (assuming the VM employs LRU page replacement order). To reduce ghost buffer statistics collection overhead, hit counts are typically maintained on segments of ghost buffer pages (*e.g.*, 4 MB) rather than on individual pages.

Our hypervisor cache-based scheme serves as an important complement to the above VM miss ratio prediction. With the hypervisor management for part of the VM memory, we can transparently trace all VM data accesses that miss the remaining VM direct memory. This allows the hypervisor to apply the ghost buffer technique to predict VM page miss rate at all memory sizes beyond the VM direct memory size (which is smaller than the currently allocated total VM memory size). In particular, this approach can predict the amount of performance loss when some memory is taken away from the VM.

4.2 Memory Allocation Policies

With known miss ratio curve for each VM at each candidate memory allocation size, we can guide multi-VM memory allocation with flexible QoS constraint and strong performance assurance. Let each VM on the host start with a baseline memory allocation, the general goal is to adjust VM memory allocation so that the overall system-wide overall page misses is reduced while certain performance isolation is maintained for each VM. Within such a context, we describe two specific allocation policies. The purpose of the first policy is to illustrate our scheme's ability in supporting flexible QoS constraint (that a sampling-based approach is not capable of). The second policy is a direct emulation of a specific sampling-based approach (employed in the VMware ESX server [23]) with an enhancement.

Isolated sharing We dynamically adjust memory allocation to the VMs with the following two objectives:

- *Profitable sharing*: Memory is divided among multiple VMs to achieve low system-wide overall page misses. In this example, we define the system-wide page miss metric as the geometric mean of each VM's miss ratio (its number of page misses under the new memory allocation divided by that under its baseline allocation). Our choice of this metric is not necessary. We should also be able to support other system-wide performance metrics as long as they can be calculated from the predicted VM miss ratio curves.
- *Isolation constraint*: If a VM's memory allocation is less than its baseline allocation, it should have a bounded performance loss (*e.g.*, no more than $\delta\%$ in additional page misses) compared to its performance under the baseline allocation.

We describe our realization of this allocation policy. One simple method is to exhaustively check all candidate allocation strategies for estimated system-wide performance metric and individual VM isolation constraint compliance. The computation overhead for such a method is typically not large for three or fewer VMs on a host. With three VMs sharing a fixed total memory size, there are two degrees of freedom in per-VM memory allocation. Assuming 100 different candidate memory sizes for each VM, around 10,000 different whole-system allocation strategies need to be checked.

The search space may become too large for exhaustive checking when there are four or more VMs on the host. In such cases, we can employ a simple greedy algorithm. At each step we try to move a unit of memory (*e.g.*, 4 MB) from the VM with the least marginal performance loss to the VM with the largest marginal performance gain if the adjustment is considered profitable (it reduces the estimated system-wide page miss metric). The VM that loses memory must still satisfy the isolation constraint at its new allocation. The algorithm stops

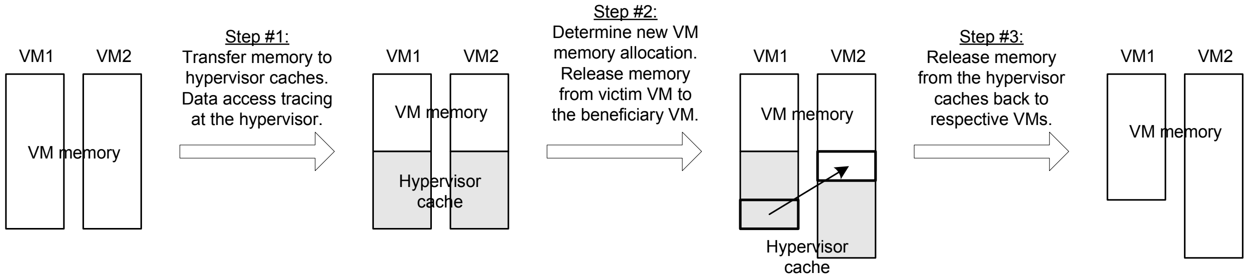

Figure 6: An example workflow of hypervisor cache-based data access tracing and multi-VM memory allocation.

when all profitable adjustments would violate the isolation constraint. Other low-overhead optimization algorithms such as simulated annealing [13] may also be applied in this case. Further exploration falls beyond the scope of this paper.

Note that in all the above algorithms, we do not physically move memory between VMs when evaluating different candidate allocation strategies. For each allocation strategy, the performance (page miss rate) of each VM can be easily estimated by checking the predicted VM page miss ratio curve.

VMware ESX server emulation The VMware ESX server employs a sampling scheme to estimate the amount of memory accessed within a period of time (*i.e.*, the working set). Based on the working set estimation, a VM whose working set is smaller than its baseline memory allocation is chosen as a *victim* — having some of its unneeded memory taken away (according to an idle memory tax parameter). Those VMs who can benefit from more memory then divide the surplus victim memory (according to certain per-VM share).

We can easily emulate the above memory allocation policy. Specifically, our per-VM page miss ratio curve prediction allows us to identify victim VMs as those whose performance does not benefit from more memory and does not degrade if a certain amount of memory is taken away. Since the VM page miss ratio curve conveys more information than the working set size alone does, our scheme can identify more victim VMs appropriately. Consider a VM that accesses a large amount of data over time but rarely reuses any of them. It would exhibit a large working set but different memory sizes would not significantly affect its page fault rate. Such problems with the working set model were well documented in earlier studies [6, 18].

4.3 An Example of Complete-System Workflow

We describe an example of complete-system workflow. We enable the hypervisor cache when a new memory allocation is desired. At such an occasion, we transfer some memory from each VM to its respective hypervisor cache and we then perform transparent data access tracing at the hypervisor. With collected traces and derived miss ratio curve for each VM, we determine a VM memory allocation toward our goal. We then release the memory from victim VM's hypervisor cache to the beneficiary VM. We finally release memory from all the hypervisor caches back to respective VMs. Figure 6 illustrates this process.

Our scheme requires a mechanism for dynamic memory adjustment between the VM memory and hypervisor cache. The ballooning technique [23] can serve this purpose. The balloon driver squeezes memory out of a VM by pinning down some memory so the VM cannot use it. The memory can be released back by popping the balloon (*i.e.*, un-pinning the memory).

The size of the hypervisor cache depends on our need of VM page miss rate information. Specifically, more complete page miss rate information (starting from a smaller candidate memory size) demands a smaller VM direct memory (and thus a larger hypervisor cache). When the predicted page miss rate information is used to guide VM memory allocation, our desired completeness of such information depends on the adjustment threshold (the maximum amount of memory we are willing to take away from the VM).

For the purpose of acquiring VM data access pattern and guiding memory allocation, the hypervisor cache can release memory back to the VM as soon as an appropriate VM memory allocation is determined. Since a new memory allocation is typically only needed once in a while, the management overhead of the hypervisor cache is amortized over a long period of time. However, if the management overhead of the hypervisor cache is not considered significant, we may keep the hypervisor cache permanently so we can quickly adjust to any new VM data access behaviors.

It should be noted that there is an inherent delay in our allocation scheme reacting to VM memory need changes — it takes a while to collect sufficient I/O access trace for predicting VM page miss ratio curve. As a result, our scheme may not be appropriate for continuously fluctuating memory re-allocations under very dynamic and adaptive VM workloads.

4.4 Vulnerability to VM Manipulation

One security concern with our adaptive memory allocation is that a selfish or malicious VM may exaggerate its memory requirement to acquire more memory allocation than needed. This might appear particularly problematic for our VM memory requirement estimation based on VM-provided page eviction information. More specifically, a selfish or malicious VM may artificially boost page eviction events so that the hypervisor would predict higher-than-actual VM memory requirement. However, we point out that the VM may achieve the same goal of exaggerating its memory requirement by artificially adding unnecessary I/O reads. Although we cannot prevent a VM from exaggerating its memory requirement, its impact on other VMs' performance is limited as long as we adhere to an appropriate performance isolation constraint.

5 Prototype Implementation

We made a proof-of-concept prototype implementation of the proposed hypervisor cache on Xen virtual machine platform (version 3.0.2) [2]. On this platform, the VMs are called xenU domains and the VM OS is a modified Linux 2.6.16 kernel. The hypervisor includes a thin core (called "hypervisor" in Xen) and a xen0 domain which runs another modified Linux 2.6.16 kernel (with more device driver support).

Our change to the xenU OS is small, mostly about notifying the hypervisor for page evictions. Since an evicted page may be transferred into the hypervisor cache, we must ensure that the page is not reused until the hypervisor finishes processing it. We achieve this by implementing the page eviction notification as a new type of I/O request. Similar to a write request, the source page will not be reused until the request returns, indicating the completion of the hypervisor processing.

The hypervisor cache and mapping tables are entirely implemented in the xen0 domain as part of the I/O backend driver. This is to keep the Xen core simple and small. The storage location in our implementation is represented by a triplet (`major device number`, `minor device number`, `block address on the device`). The cache is organized in a queue of pages. Both the cache and mapping tables are indexed with hash tables to speed up the lookup. Our hypervisor cache supports both delayed writes and write-through.

The main purpose of our prototype implementation is to demonstrate the correctness of our design and to illustrate the effectiveness of its intended utilization. At the time of this writing, our implementation is not yet fully optimized. In particular, we use explicit page copying when transferring data between the VM memory and the hypervisor cache. A page remapping technique is used in Xen to pass incoming network packet from the privileged driver domain (xen0) to a normal VM (xenU). However, our measured cost for this page remapping technique does not exhibit significant advantage compared to the explicit page copying, which is also reported in an earlier study [11].

Primitive operations	*Overhead*
Mapping table lookup	0.28 μs
Mapping table insert	0.06 μs
Mapping table delete	0.06 μs
Cache lookup	0.28 μs
Cache insert (excl. page copying)	0.13 μs
Cache delete	0.06 μs
Cache move to tail	0.05 μs
Page copying	7.82 μs

Table 1: Overhead of primitive cache management operations on a Xeon 2.0 GHz processor.

Table 1 lists the overhead of primitive cache management operations on a Xeon 2.0 GHz processor. Each higher-level function (read cache hit, read cache miss, write cache hit, write cache miss, eviction cache hit, and eviction cache miss) is simply the combination of several primitive operations. Page copying is the dominant cost for read cache hit and eviction cache miss. The cost for other functions is within 1 μs.

6 Evaluation

We perform experimental evaluation on our prototype hypervisor cache. The purpose of our experiments is to validate the correctness of our cache design and implementation (Section 6.2), evaluate its performance and management overhead (Section 6.3), validate its VM miss ratio curve prediction (Section 6.4), and demonstrate its effectiveness in supporting multi-VM memory allocation with flexible QoS objectives (Section 6.5). The experimental platform consists of machines each with one 2.0 GHz Intel Xeon processor, 2 GB of physical memory, and two IBM 10 KRPM SCSI drives.

6.1 Evaluation Workloads

Our evaluation workloads include a set of microbenchmarks and realistic applications/benchmarks with significant data accesses. All workloads are in the style of on-demand services.

Microbenchmarks allow us to examine system behaviors for services of specifically chosen data access patterns. All microbenchmarks we use access a dataset of 500 4 MB disk-resident files. On the arrival of each request, the service daemon spawns a thread to process it. We employ four microbenchmarks with different data access patterns:

- *Sequential*: We sequentially scan through all files one by one. We repeat the sequential scan after all files are accessed. Under LRU page replacement, this access pattern should result in no cache hit when the memory size is smaller than the total data size.
- *Random*: We access files randomly with uniform randomness — each file has an equal probability of being chosen for each access.
- *Zipf*: We access files randomly with a Zipf distribution — file i ($1 \leq i \leq 500$) is accessed with a probability proportional to $\frac{1}{i^{\alpha}}$. The exponent $\alpha = 1.0$ in our test.
- *Class*: We divide files into two classes: one tenth of all files are in the popular class and the rest are in the normal class. Each file in the popular class is 10 times more likely to be accessed than each file in the normal class.

Each microbenchmark also has an adjustable write ratio. The write ratio indicates the probability for each file access to be a write. A read file access reads the entire file content in 64 KB chunks. A write file access overwrites the file with new content of the same size. Writes are also performed in 64 KB chunks.

In addition to the microbenchmarks, our experiments also include four realistic data-intensive services.

- *SPECweb99:* We implemented the static content portion of the SPECweb99 benchmark [19] using the Apache 2.0.44 Web server. This workload contains 4 classes of files with sizes at 1 KB, 10 KB, 100 KB, and 1,000 KB respectively and the total dataset size is 4.9 GB. During each run, the four classes of files are accessed according to a distribution that favors small files. Within each class, a Zipf distribution with exponent $\alpha = 1.0$ is used to access individual files.
- *Index searching:* We acquired a prototype of the index searching server and a dataset from the Web search engine Ask Jeeves [1]. The dataset contains the search index for about 400,000 Web pages. It includes a 66 MB mapping file that maps MD5-encoded keywords to proper locations in the search index of 2.4 GB. For each keyword in an input query, a binary search is first performed on the mapping file and then search indexes of query keywords are then accessed. The search query words in our test workload are based on a one-week trace recorded at the Ask Jeeves site in 2002.
- *TPC-C:* We include a local implementation of the TPC-C online transaction processing benchmark [20]. TPC-C simulates a population of terminal operators executing Order-Entry transactions against a database. In our experiments, the TPC-C benchmark runs on the MySQL 5.0.18 database with a database size of 2.6 GB.
- *TPC-H:* We evaluate a local implementation of the TPC-H decision support benchmark [21]. The TPC-H workload consists of 22 complex SQL queries. Some queries require excessive amount of time to finish and they are not appropriate for interactive on-demand services. We choose a subset of 17 queries in our experimentation: Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q11, Q12, Q13, Q14, Q15, Q17, Q19, Q20, and Q22. In our experiments, the TPC-H benchmark runs on the MySQL 5.0.18 database with a database size of 496 MB.

6.2 Cache Correctness

We augment the microbenchmarks to check the correctness of returned data from the hypervisor cache. We do so by maintaining a distinct signature for each 1 KB block of each file during the file creations and overwrites. Each file read access checks the signatures of returned content, which would fail if the content were incorrectly cached. We tested the four microbenchmarks at three different write ratios (0%, 10%, and 50%) and a variety of VM memory and hypervisor cache sizes. We found no signature checking failures over all test runs.

We also ran tests to check the necessity of our cache correctness support described in Section 3.2. We changed the hypervisor so that it does not capture all page eviction/release or that it does not check the reverse mapping from storage locations to VM memory pages when admitting evicted data. We detect incorrect content for both cases and we traced the problems to the error cases described in Section 3.2.

6.3 Performance and Management Overhead

We evaluate the cache performance and management overhead of our hypervisor exclusive cache. For each workload, we configure the total available memory (combined size of the VM memory and hypervisor cache) to be 512 MB. In the baseline scheme, all memory is directly managed by the VM and there is no hypervisor cache. We then examine the cases when we transfer 12.5%, 25%, 50%, and 75% of the memory (or 64 MB, 128 MB, 256 MB, and 384 MB respectively) to be managed by the hypervisor cache. Note that some setting ("75% memory to cache") may not be typical in practice. Our intention is to consider a wide range of conditions in this evaluation.

We look at three performance and overhead metrics: the overall service request throughput (Figure 7), the I/O overhead per request (Figure 8), and CPU overhead in hypervisor cache management (Figure 9). Here the I/O overhead only counts those page I/Os that reach the real

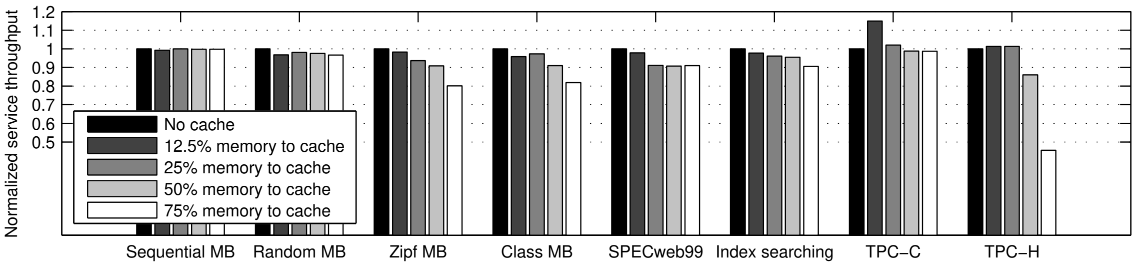

Figure 7: Service request throughput of different hypervisor caching schemes normalized to that of no cache.

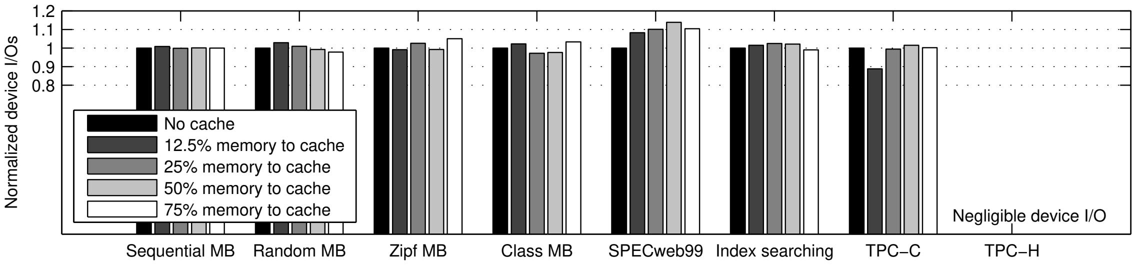

Figure 8: The I/O overhead per request of different schemes normalized to that of no cache.

storage device (*i.e.*, those that miss both VM memory and the hypervisor cache). Note that since we intend to compare the overhead when finishing the same amount of work, the page I/O overhead per request (or workload unit) is a better metric than the page I/O overhead per time unit. In particular, two systems may exhibit the same page I/O overhead per time unit simply because they are both bound by the maximum I/O device throughput. For the same reason, we use a scaled CPU overhead metric. The scaling ratio is the throughput under "no cache" divided by the throughput under the current scheme.

Among the eight workloads in our experimental setup, TPC-H is unique in the sense that it is completely CPU-bound with 512 MB memory. Below we analyze the results separately for it and the other workloads.

Seven non-CPU-bound workloads. In terms of service throughput, the degradation compared to "no cache" is less than 20% in all cases and no more than 9% excluding the extreme condition of "75% memory to cache". This is largely because the employment of our hypervisor cache does not significantly increase the system I/O overhead (as shown in Figure 8). A closer examination discovers an I/O overhead increase of up to 13% for SPECweb99. This is because our hypervisor cache does not cache evicted VM data that is not in OS page buffer, such as file meta-data like `inode` and `dentry` in Linux. Most files accessed by SPECweb99 are very small and thus the effect of not caching file meta-data is more pronounced. Note that the hypervisor caching of file meta-data requires the understanding of file meta-data memory layout, which would severely compromise the transparency of the hypervisor cache. Excluding SPECweb99, the page fault rate varies between a 11% decrease and a 5% increase compared to "no cache" over all test cases. Now we consider the CPU overhead incurred by cache management and minor page faults (shown in Figure 9). Overall, the additional overhead (compared to the "no cache" case) is up to 6.7% in all cases and up to 3.2% excluding the extreme condition of "75% memory to cache". Its impact on the performance of non-CPU-bound workloads is not substantial.

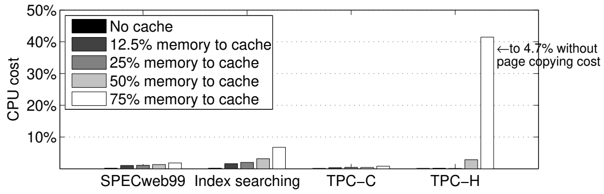

Figure 9: CPU cost of hypervisor cache for four real services. We do not show results for microbenchmarks since they do not contain any realistic CPU workload.

CPU-bound TPC-H. There is no real device I/O overhead in all test cases and its performance difference is mainly determined by the amount of additional CPU overhead of the cache management. Such cost is negligible for "12.5% memory to cache" and "25% memory to cache". It is more significant for "50% memory to cache" and "75% memory to cache", causing 14% and 54% throughput degradation respectively compared to "no cache". This is largely due to the costly page copying operations. Excluding the page copying overhead, the expected CPU overhead at "75% memory to cache" would be reduced from 41% to 4.7%. This indicates that

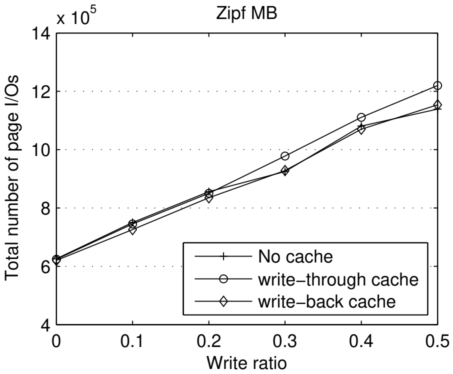

Figure 10: Performance impact of write-through/write-back caches in a system with 256 MB VM direct memory and 256 MB hypervisor cache. The y-axis shows the total number of page I/Os during the Zipf test with 2,000 file requests at different write ratios.

the concern on CPU cost can be significantly alleviated with an efficient page transfer mechanism.

As we discussed in Section 3.1, employing write-through at the hypervisor cache maintains the persistence semantic of the write completion. However, write-through is not as effective as delayed writes in caching the write I/O traffic. To illustrate the performance implication of two write strategies employed by the hypervisor cache, we run the Zipf microbenchmark with different write ratios ranging from 0 (read-only) to 0.5. The result in Figure 10 indicates that write-through indeed yields more I/O operations than the original system (around 7% at 0.5 write ratio), whereas delayed writes does not increase the number of page I/Os.

We summarize the performance and overhead results as follows. The employment of hypervisor cache does not increase the system I/O overhead (excluding an exceptional case). The CPU overhead for our current prototype implementation can be significant, particularly at the extreme setting of "75% memory to cache". However, our results suggest that the CPU overhead does not have large impact on the performance of services that are not CPU-bound. We also expect that a more optimized cache implementation in the future may reduce the CPU cost.

6.4 Accuracy of Miss Ratio Curve Prediction

We perform experiments to validate the accuracy of our VM miss ratio curve (page miss rate *vs.* memory size curve) prediction. Jones *et al.* [12] have demonstrated the prediction of VM miss ratio curve for memory sizes larger than the current allocation. The contribution of our hypervisor cache-based transparent data access tracing is to predict VM miss ratio curve for memory sizes smaller than the current allocation. In practice, we predict a miss ratio curve that includes memory sizes both larger and smaller than the current allocation. Such a curve can tell the performance degradation when the memory allocation is reduced as well as the performance improvement when the memory allocation is increased. Both pieces of information are necessary to determine the VM memory allocation with performance assurance.

We use a system configuration with a large hypervisor cache to produce the VM miss ratio curve over a wide range. With a memory allocation of 512 MB, 384 MB is managed as the hypervisor cache and the VM memory has 128 MB left. This setting allows us to predict the VM miss ratio curve from the memory size of 128 MB. Smaller hypervisor caches may be employed in practice if we have a bound on the maximum amount of VM memory reduction, or if the management overhead for a large cache is considered too excessive. We validate the prediction accuracy by comparing against measured miss ratios at several chosen memory sizes. The validation measurements are performed on VM-only systems with no hypervisor cache.

Figure 11 illustrates the prediction accuracy for the eight workloads over memory sizes between 128 MB and 1024 MB. Results suggest that our prediction error is less than 15% in all validation cases. Further, the error is less than 9% for memory sizes smaller than the current allocation (512 MB), which is the primary target of our hypervisor cache-based miss ratio prediction. We believe the prediction error is due to the imperfect LRU replacement employed in the VM OS.

Since we know the microbenchmark data access patterns, we can also validate their miss ratio curves with simple analysis. Sequential MB has a flat curve since there can be no memory hit as long as the memory size is less than the total data size. Random MB's data access miss rate should be $1 - \frac{\text{memory size}}{\text{data size}}$ and therefore its miss ratio curve is linear. Zipf MB and Class MB have more skewed data access patterns than Random MB so the slopes of their miss ratio curves are steeper.

6.5 Multi-VM Memory Allocation

Guided by the predicted VM miss ratio curves, we perform experiments on multi-VM memory allocation with performance assurance. Our experiments are conducted with the allocation goals of *isolated sharing* and *VMware ESX server emulation* described in Section 4.2 respectively. In our experiments, we employ three VMs, running SPECweb99, index searching, and TPC-H respectively. The initial baseline memory allocation for each VM is 512 MB. We adjust the total 1,536 MB memory among the three VMs toward our allocation goal.

Isolated sharing In isolated sharing experiments, we attempt to minimize a system-wide page miss metric

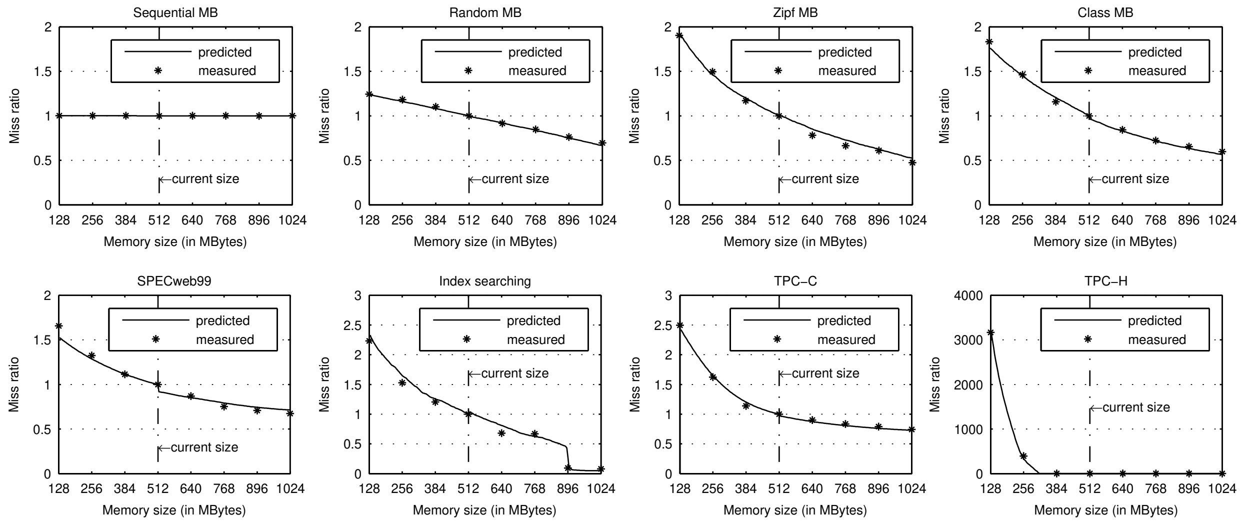

Figure 11: Accuracy of miss ratio curve prediction for memory sizes between 128 MB and 1024 MB. The current memory allocation is 512 MB, within which 384 MB is managed by the hypervisor cache. The miss ratio at each memory size is defined as the number of page misses at the current memory size divided by the page miss number at a baseline memory size (512 MB in this case). Since there is almost no page miss for TPC-H at 512 MB memory, we set a small baseline page miss number for this workload to avoid numeric imprecision in divisions.

(*e.g.*, profitable sharing) while at the same time no VM should experience a performance loss beyond a given bound (*i.e.*, isolation constraint). In our experiments, the system-wide page miss metric is the geometric mean of all VMs' miss ratios which represents the average acceleration ratio for all VMs in terms of page miss reduction. The performance loss bound is set as a maximum percentage increase of page misses compared to the baseline allocation. To demonstrate the flexibility of our policy, we run two experiments with different isolation constraints — 5% and 25% performance loss bounds respectively. We allocate memory in the multiple of 4 MB. We use exhaustive search to find the optimal allocation strategy and the computation overhead for the exhaustive search is acceptable for three VMs.

Table 2 lists the memory allocation results. Overall, the experimental results show that our hypervisor cache-based memory allocation scheme can substantially reduce the system-wide page miss metric (15% average page miss reduction at 5% isolation constraint and 59% average page miss reduction at 25% isolation constraint). This is primarily due to our ability of transparent data access tracing and accurate VM miss ratio curve prediction. The two very different allocation outcomes at different isolation constraints demonstrate the flexibility and performance assurance of our approach. In comparison, a simple working set-based allocation approach [23] may not provide such support.

Generally all VMs observe the isolation constraints in our experiments. However, a small violation is observed for SPECweb99 in the test with 25% isolation constraint (28% page miss increase). This is due to the miss ratio curve prediction error. We believe such a small violation is tolerable in practice. If not, we can leave an error margin when determining the allocation (*e.g.*, using a 20% isolation constraint on the predicted miss ratios when a hard 25% isolation constraint needs to be satisfied).

Initial configuration

	VM #1	VM #2	VM #3
Workload	SPECweb	Searching	TPC-H
Memory alloc.	512 MB	512 MB	512 MB

Allocation with 5% isolation constraint

	VM #1	VM #2	VM #3
Memory alloc.	452 MB	748 MB	336 MB
Predicted miss ratio	1.05	0.64	1.00
Predicted geo. mean	0.88		
Measured miss ratio	1.04	0.58	1.00
Measured geo. mean	0.85		

Allocation with 25% isolation constraint

	VM #1	VM #2	VM #3
Memory alloc.	280 MB	920 MB	336 MB
Predicted miss ratio	1.24	0.06	1.00
Predicted geo. mean	0.43		
Measured miss ratio	1.28	0.05	1.00
Measured geo. mean	0.41		

Table 2: Memory allocation results for isolated sharing.

VMware ESX server emulation This experiment demonstrates that hypervisor cache-based allocation is

Initial configuration

	VM #1	VM #2	VM #3
Workload	SPECweb	Searching	TPC-H
Memory alloc.	512 MB	512 MB	512 MB

Hcache emulation

	VM #1	VM #2	VM #3
Memory alloc.	600 MB	600 MB	336 MB
Measured miss ratio	0.85	0.71	1.00

VMware ESX server

	VM #1	VM #2	VM #3
Memory alloc.	576 MB	576 MB	384 MB
Measured miss ratio	0.88	0.78	1.00

Hcache emulation (A background task runs with TPC-H)

	VM #1	VM #2	VM #3
Memory alloc.	578 MB	578 MB	380 MB
Measured miss ratio	0.88	0.78	1.00

VMware ESX server (A background task runs with TPC-H)

	VM #1	VM #2	VM #3
Memory alloc.	512 MB	512 MB	512 MB
Measured miss ratio	1.00	1.00	1.00

Table 3: Memory allocation results for VMware ESX server emulation.

able to: 1) emulate the memory allocation policy employed in VMware ESX server; and 2) more accurately discover VM memory need when choosing victim VMs. We employ VMware ESX server version 3.0.0 in this test. We ported our test workloads to ESX server environment and conditioned all workload parameters in the same way. For each VM, the initial and maximum memory allocations are 512 MB and 1 GB respectively. All VMs receive the same share (a VMware ESX server parameter indicating a per-VM proportional right to memory).

We first use the exactly same three VMs as in the isolated sharing experiments. Table 3 shows that both ESX server and our emulation are able to reclaim unused memory from TPC-H VM without raising the page fault rate. However, ESX server is more conservative, resulting in less performance improvement for other VMs. We suspect this conservatism is related to the prediction inaccuracy inherent with its memory sampling approach.

Then we add a light background workload to TPC-H VM. The workload touches one 4 MB file every 2.5 seconds over 500 such files repeatedly. Figure 12 shows the predicted and measured miss ratio curve for this new TPC-H VM. It is clear that beyond the allocation of around 360 MB, more memory does not reduce the VM page fault rate. Our hypervisor cache-based allocation

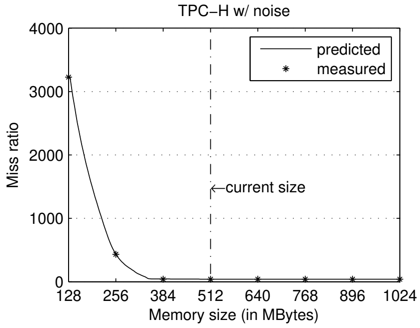

Figure 12: Miss ratio curve of TPC-H with a light background task.

correctly recognizes this and therefore takes away some TPC-H VM memory to the other two VMs. In contrast, ESX server estimates that the TPC-H VM has a large working set (around 830 MB) and thus does not perform any memory allocation adjustment. This is due to inherent weakness of the working set model-based memory allocation [6, 18].

7 Conclusion

For data-intensive services on a virtual machine (VM) platform, the knowledge of VM page misses under different memory resource provisioning is desirable for determining appropriate VM memory allocation and for facilitating service consolidation. In this paper, we demonstrate that the employment of a hypervisor-level exclusive buffer cache can allow transparent data access tracing and accurate prediction of the VM page miss ratio curve without incurring significant overhead (no I/O overhead and mostly small CPU cost). To achieve this goal, we propose the design of the hypervisor exclusive cache and address challenges in guaranteeing the cache content correctness when the data enters the cache directly from the VM memory.

As far as we know, existing hypervisor-level buffer cache is used primarily for the purpose of keeping single copy of data shared across multiple VMs. Our hypervisor exclusive cache is unique in its ability to manage large chunk of a VM's memory without increasing the overall system page faults. Although our utilization of this cache is limited to transparent data access tracing in this paper, there might also be other beneficial use of the cache. For example, the hypervisor-level buffer cache allows the employment of new cache replacement policy and I/O prefetching policy transparent to the VM OS. This may be desirable when the OS-level caching and I/O prefetching are not fully functional (*e.g.*, during OS installation or boot [10]) or when the default OS-level policy is insufficient (*e.g.*, desiring more aggressive I/O prefetching [14]).

Acknowledgments We would like to thank Irfan Ahmad (VMware Inc.) for helpful discussions in the preparation of this work. We are also grateful to the USENIX anonymous reviewers for their useful comments that improved this paper.

References

[1] Ask jeeves search. http://www.ask.com.

[2] 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 *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, October 2003.

[3] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. on Computer Systems*, 15(4):412–447, November 1997.

[4] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The collective: A cache-based system management architecture. In *Proc. of the Second USENIX Symp. on Networked Systems Design and Implementation*, pages 259–272, Boston, MA, May 2005.

[5] Z. Chen, Y. Zhou, and K. Li. Eviction based placement for storage caches. In *Proc. of the USENIX Annual Technical Conf.*, pages 269–282, San Antonio, TX, June 2003.

[6] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational databases systems. In *Proc. of the 11th Int'l Conf. on Very Large Data Bases*, pages 127–141, Stockholm, Sweden, August 1985.

[7] C. Clark, K. Fraser, S. Hand, J. J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of the Second USENIX Symp. on Networked Systems Design and Implementation*, pages 273–286, Boston, MA, May 2005.

[8] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent, 6397242, October 1998.

[9] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, pages 154–169, Kiawah Island, SC, December 1999.

[10] I. Ahmad, VMware Inc., July 2006. Personal communication.

[11] Y. Turner J. R. Santos, G. J. Janakiraman. Network optimizations for PV guests. Xen Summit, September 2006. http://www.xensource.com/files/summit_3 /networkoptimizations.pdf.

[12] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 13–23, San Jose, CA, October 2006.

[13] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[14] C. Li, K. Shen, and A. E. Papathanasiou. Competitive prefetching for concurrent sequential I/O. In *Proc. of the Second EuroSys Conf.*, pages 189–202, Lisbon, Portugal, March 2007.

[15] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[16] M. K. McKusick. Disks from the perspective of a file system. *USENIX ;login:*, 31(3):18–19, June 2006.

[17] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 79–95, Copper Mountain Resort, CO, December 1995.

[18] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. on Database Systems*, 11(4):473–498, December 1986.

[19] SPECweb99 benchmark. http://www.specbench.org /osg/web99.

[20] Transaction Processing Performance Council. TPC benchmark C. http://www.tpc.org/tpcc.

[21] Transaction Processing Performance Council. TPC benchmark H. http://www.tpc.org/tpch.

[22] VMware Infrastructure - ESX Server. http://www .vmware.com/products/esx.

[23] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, pages 181–194, Boston, MA, December 2002.

[24] M. Williamson. Xen wiki: XenFS. http://wiki.xensource .com/xenwiki/XenFS.

[25] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proc. of the USENIX Annual Technical Conf.*, pages 161–175, Monterey, CA, June 2002.

[26] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*, pages 103–116, Seattle, WA, November 2006.

[27] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 177–188, Boston, MA, October 2004.