Synergy: A Hypervisor Managed Holistic Caching System

Debadatta Mishra, Purushottam Kulkarni, and Raju Rangaswami

Abstract—Efficient system-wide memory management is an important challenge for over-commitment based hosting in virtualized systems. Due to the limitation of memory domains considered for sharing, current deduplication solutions simply cannot achieve system-wide deduplication. Popular memory management techniques like sharing and ballooning enable important memory usage optimizations individually. However, they do not complement each other and, in fact, may degrade individual benefits when combined. We propose Synergy, a hypervisor managed caching system to improve memory efficiency in over-commitment scenarios. Synergy builds on an exclusive caching framework to achieve, for the first time, system-wide memory deduplication. Synergy also enables the co-existence of the mutually agnostic ballooning and sharing techniques within hypervisor managed systems. Finally, Synergy implements a novel file-level eviction policy that prevents hypervisor caching benefits from being squandered away due to partial cache hits. Synergy's cache is flexible with configuration knobs for cache sizing and data storage options, and a utility-based cache partitioning scheme. Our evaluation shows that Synergy consistently uses 10% to 75% lesser memory by exploiting system-wide deduplication as compared to inclusive caching techniques and achieves application speedup of 2x to 23x. We also demonstrate the capabilities of Synergy to increase VM packing density and support for dynamic reconfiguration of cache partitioning policies.

Index Terms—Virtualization, Memory Virtualization, Resource Management.

1 Introduction

Memory in virtualized systems is a precious resource. The *average* utilization rates for memory are reported to be the highest by a large margin (40% compared to less than 6% utilization of the other resource types) in production ESX servers [1]. Memory over-commitment is a popular technique for enhancing memory utilization and to increase packing density of virtual machines. Over-commitment relies on temporal non-overlapping memory demands and effective statistical multiplexing of the resource.

The mechanisms available for facilitating memory over-commitment are: *(i) ballooning* [2], [3], a guest intrusive memory provisioning mechanism, *(ii) demand paging* [3], a hypervisor based page swapping mechanism, *(iii) sharing* [4], [5], [6], a hypervisor mechanism to deduplicate similar pages, and *(iv) hypervisor caching* [7], [8], [9], a hypervisor store for efficient storage of pages. Of these, sharing is the only technique that has the potential to drastically reduce VM memory footprint (and hence improve memory efficiency) with little performance overhead for VM applications. Previous studies have shown that in-memory disk caches offer significant sharing opportunities [4], [5], [6]. Additionally, in virtualized environments, disk content can be cached by the hypervisor and be used as a *second-chance* cache. Hypervisor caches (e.g., Linux *transcendent memory* [8], [9]) are a system-wide solution to increase memory efficiency. The hypervisor cache stores *clean* pages evicted from VMs to provide exclusive caching with ephemeral storage semantics (explained in §3.1). Furthermore, this cache can be compressed, deduplicated or subjected to per-VM partitioning and eviction policies. Thus, several solution configurations become available with hypervisor caches. Unfortunately, the available memory management mechanisms— deduplication-based sharing, ballooning and hypervisor caching do not complement each other and are not designed to cooperatively explore the solution space for improving memory efficiency. The current juxtaposition of these techniques suffers a few significant drawbacks.

First, while deduplicating the hypervisor cache [8], and sharing memory pages within VMs [10], [11], [3] are both individually possible, system-wide sharing of all of disk pages is not. Out-of-band sharing solutions (memory scan-and-merge) [10], [3] only consider anonymous memory for deduplication and do not apply to hypervisor managed pages. In-band sharing techniques (intercept-and-merge memory accesses) [12], [13] or redundancy elimination in cache hierarchies, as employed by systems like Singleton [14], do not account for pages stored in the hypervisor cache either. Further, these techniques do not allow straightforward extensions that include hypervisor managed caches. As a result, there exists no current solution that performs system-wide deduplication on pages of the virtual machine, the hypervisor cache and the hypervisor.

Second, ballooning plays an important role of exerting memory pressure on the guest OS disk cache and gives adequate control to the hypervisor in managing system-wide memory allocation. While it is crucial for ballooning and sharing to co-exist to provide efficient exclusive caching, these techniques are not compatible. Past techniques [3] have employed a non-overlapping sequence of operations for these techniques—employ ballooning first and then sharing, strictly in this order. Such sequencing is obviously very restrictive for adaptive and dynamic management of memory as demand and virtual machine packing density changes. Further, ballooning out (evicting) pages from inside virtual machines that are shared across virtual machines does not increase free memory in the system. Moreover, shared pages that get ballooned out

- *D. Mishra and P. Kulkarni are with the Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India. E-mail: deba,puru@cse.iitb.ac.in*
- *R. Rangasami is with Florida International University. Email: raju@cs.fiu.edu*

(evicted due to ballooning), break sharing and thus negatively impact memory efficiency. Later in §2.2, we illustrate loss of sharing due to ballooning and resultant memory capacity overhead of as much as 65%.

Finally, hypervisor exclusive caches do not complement guest disk caches sufficiently today because the eviction logic at the hypervisor level is oblivious of guest page cache usage. For example, given the disk block prefetch (or read-ahead) [15], [16] implemented within guest OSs, retaining only a subset of the prefetched blocks in the hypervisor cache is hardly useful. Fetching the remaining blocks incurs disk I/O and does not benefit from the cache hits.

Towards addressing these drawbacks of memory over-commitment techniques, main contributions of our work are,

- Design and implementation of Synergy, an optimized hypervisor managed caching solution that provides comprehensive *system-wide* deduplication of all disk cache pages used by virtual machines and a mutually-exclusive hypervisor cache. Synergy, also provides cache configuration flexibility in terms of per-VM cache sizing and storage options for cached objects (uncompressed or compressed).
- Incorporate within the design of Synergy *seamless co-existence of sharing and ballooning* while providing high memory utilization.
- Demonstrate inefficiency of caching a subset of prefetched blocks in the hypervisor cases and developing a novel file granularity based eviction policy towards increase cache usage efficiency.

We perform extensive empirical evaluation of Synergy using a variety of read-intensive and write-intensive workloads. We evaluate different caching modes enabled by Synergy vis-a-vis other existing caching solutions. Our results show that Synergy consistently uses 10% to 75% lesser memory by exploiting system-wide deduplication as compared to inclusive caching. We demonstrate that Synergy enables improved memory management in over-commitment based cloud hosting solutions. Synergy's improved memory efficiency results in tighter packing of VMs compared to other exclusive caching solutions like Singleton [14] and zcache [17]. Further, we demonstrate usage of Synergy cache provisioning framework to guide design of policies for effective use of the hypervisor managed cache. Finally, we illustrate a holistic memory management policy using Synergy features to show the effectiveness of Synergy in a memory over-commit scenario.

2 MOTIVATION

As part of their memory management subsystem, hypervisors have evolved to adopt ballooning, deduplication-based sharing, and hypervisor caching. Unfortunately, given the relative independence of memory management across the virtual machine and hypervisor layers, these optimizations do not all work well together in memory over-committed setups. In this section, we discuss memory management gaps in virtualized systems that motivate each of our contributions.

2.1 System-wide memory deduplication

Out-of-band (offline) scanning techniques for content based sharing such KSM (Kernel Samepage Merging) [10], [18], [3] only consider pages that are *anonymous* and not those stored by the hypervisor cache. On the other hand, hypervisor managed caches [8] provide a central store for caching disk content across VMs and can be used to improve memory utilization via cross-VM deduplication, cache partitioning on a per-VM basis and compression of cache contents. However, these techniques do not consider pages allocated to VMs for deduplication. System-wide memory deduplication requires a unified approach that addresses all memory usages. Since KSM sharing is out-of-band, in-band transfers of pages between a VM and hypervisor (as part of hypervisor caching) does not invoke the necessary copy-on-write (CoW) protection mechanisms for overwriting shared pages. This imposes additional challenges for simultaneous employment and cooperation of the two techniques for system-wide deduplication. Further, while techniques like Singleton [14] provide system-wide exclusivity, they do not provide system-wide deduplication. The Singleton approach drops common disk cache pages from the hypervisor cache, but does not deduplicate content in the hypervisor cache.

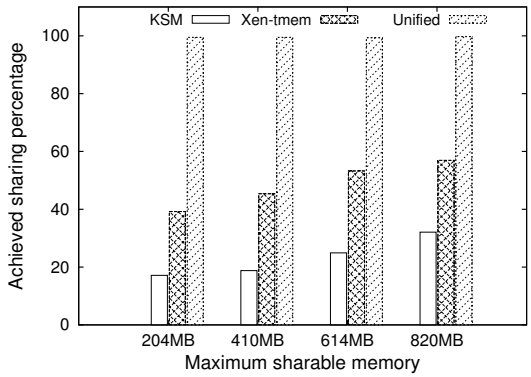


Fig. 1: Sharing potential realization with existing sharing techniques vs. the proposed unified approach.

To contrast these approaches with an unified approach that implements system-wide deduplication, we performed an experiment wherein a file of size 1 GB is read from within a VM of size 512 MB and achieved system-wide sharing is measured. Sharable content within the file is varied. For each run, the measured number of pages that are deduplicated after completion of the file read is shown in Figure 1. Xen-`tmem` (the hypervisor caching solution in Xen) resulted in higher sharing (∼50% of the potential) compared to KSM, but did not achieve the full potential because of the limited sharing domain. An unified approach achieves 100% sharing and illustrates the potential of system-wide memory deduplication across all virtual machines and the hypervisor managed cache.

2.2 Ballooning with sharing

Ballooning provides the important ability to dynamically resize VM memory and indirectly the guest OS (disk) page cache. However, current out-of-band scanning-based sharing solutions such as KSM and guest balloon modules are agnostic to each other. This may lead to a situation where shared memory pages (not consuming any extra memory) are ballooned out. As part of the CoW semantics, a shared page to be evicted has to be assigned a separate machine page. Simultaneously, a page will be reclaimed by the hypervisor as part of the balloon inflation process—a zero reclamation benefit. This breaks sharing, provides

Balloon size	Reclaimed memory (KSM OFF)	Reclaimed memory (KSM ON)	Shared memory (KSM ON)
0 MB	0 MB	0 MB	455 MB
200 MB	200 MB	35 MB	333 MB
400 MB	400 MB	122 MB	205 MB
600 MB	600 MB	216 MB	110 MB

TABLE 1: Non-complimentary behavior of ballooning and sharing.

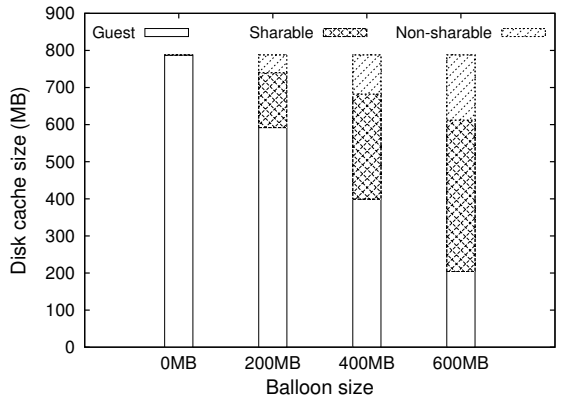


Fig. 2: Impact on guest disk cache with ballooning.

no memory reclamation benefits, and thereby adversely impacts memory utilization.

We performed an experiment wherein a file of size 1GB (sharable content of 614MB) is read from a VM (configured with 1GB memory) followed by balloon inflation to reclaim memory. Table 1 shows the amount of memory reclaimed and the level of sharing for different balloon sizes. As several shared pages are broken, reclaimed memory size does not match balloon size. Memory sharing due to KSM decreases by 75% and reclaimed memory mismatch is of ∼65% with a balloon size of 600 MB.

On the other hand, ballooning can be quite valuable for holistic memory caching; if a VM does not benefit from disk caching, the corresponding memory can be ballooned out and subsequently used by a hypervisor managed cache to store the disk contents of other VMs. A system-wide sharing solution that spans the boundaries of the hypervisor managed cache and the guest memory can address this need well. *Shared disk cache pages evicted by VMs that enter the hypervisor cache can continue to be shared without consuming any extra memory.*

For the same experiment mentioned earlier, Figure 2 demonstrates that as the guest disk cache size decreases (approximately equal to the ballooned memory size), an unified sharing scheme will be able to re-share a significant "Sharable" portion of the evicted memory. Only the "Non-sharable" pages have to be explicitly stored by the hypervisor to achieve similar caching performance as when no ballooning (Balloon size = 0) is used. A technique that allows ballooning and sharing to co-exist and retain benefits of sharing can thus significantly improve memory efficiency.

2.3 Read-ahead aware eviction policy

Modern operating systems perform sequential prefetching of file data from disk, also known as *read-ahead* [15], [16], to reduce data access latency for future file operations. Given the widespread adoption of read-ahead, eviction algorithms of hypervisor based second chance caches should complement the read-ahead optimizations of the guest OS. Table 2 shows cache hit ratios of the KVM-`tmem` hypervisor cache [17] with its default eviction policy and a read-ahead aware policy (discussed in §5). The Filebench `webserver` benchmark was executed with different number of application threads. With increased number of application I/O threads, the cache hit ratio with the default KVM-`tmem` eviction algorithm decreased significantly from 69% for 4 threads to as low as 1% for 32 threads. On the other hand, a read-ahead aware eviction algorithm yielded significantly higher cache hit ratios (91% for 4 threads to 66% for up to 32 threads). A read-ahead aware eviction policy to manage hypervisor caches is thus vital, especially for workloads with significant sequential access patterns.

# of IO threads	Cache hit ratio (%) (KVM-`tmem` eviction)	Cache hit ratio (%) (RA aware eviction)
4	69	91
8	64	88
16	34	80
32	1	66

TABLE 2: Cache hit improvement of read-ahead aware eviction policy for hypervisor caches.

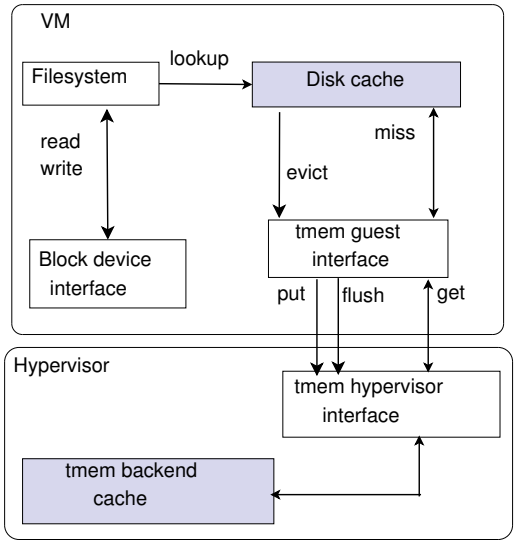


Fig. 3: High-level architecture of the `tmem` hypervisor caching system.

3 The Synergy Caching Solution

Design of Synergy is based on the premise of integrating benefits of the three memory management techniques—ballooning, hypervisor caching and sharing. As part of this section, we first provide a background of hypervisor caching and then present design details of Synergy.

3.1 Hypervisor caching

Hypervisor caching aims to increase the system-wide memory efficiency in virtualized environments. One usage scenario is to export the page cache of all VMs to the hypervisor cache. Towards improving memory efficiency, the central hypervisor cache can deduplicate and compress objects across VMs and also provide

Feature	Xen-tmem	Desired
Extent of deduplication	Tmem store is deduplicated	System-wide deduplication
Co-existence of ballooning and sharing	Unsupported	Ballooned Pages are re-shared
Eviction policy	Agnostic to guest policy	Guest OS read-ahead aware

TABLE 3: Shortcomings of Xen-tmem hypervisor cache to facilitate desired holistic caching.

policies of cache partitioning and per-VM eviction policies. Another scenario is to use the hypervisor cache opportunistically as a second-chance cache. Objects evicted from VMs are stored in the hypervisor cache whenever hypervisor has memory to spare and the above mentioned optimizations are applied. Design of the Synergy caching solution is amenable to both usage scenarios.

Transcendent memory (or `tmem`) [8] is a state-of-the-art hypervisor-managed cache which implements exclusive caching by design [9]. The `tmem` cache stores *clean* disk pages and thus is transparent to storage durability mechanisms. `Tmem` (illustrated in Figure 3) has three major components—(i) a guest `tmem` interface, (ii) a `tmem` back-end cache, and (iii) a `tmem` hypervisor interface to access the cache.

On a disk page lookup failure inside the VM, the guest OS looks up the page in the `tmem` cache using a `get` operation. Upon evicting a clean (i.e., unmodified) page from its cache, the guest OS stores the page in the `tmem` cache using a `put` operation. Upon updating a disk page, the guest issues a `flush` operation that purges the page from the `tmem` cache, if present, since the `tmem` version of the page is now outdated. The above operations ensure, exclusive caching between the guest disk cache and the `tmem` cache (on a per VM basis). The `tmem` cache provides a key-value interface for storing page objects. Typically, the key is a combination of the inode number of the file and the block offset of the page within the file. The value is the page being inserted or looked-up. Further, `tmem` provides *ephemeral* semantics for cached objects, i.e., a page `put` in the cache is not guaranteed to be returned on a `get`, since it may have been evicted from the `tmem` cache.

Current hypervisor caching solutions [8], [17] fail to provide an integrated memory management platform in several aspects. Shortcomings of the state-of-the-art Xen `tmem` solution are presented in Table 3.

3.2 Cache design

Traditional deduplication mechanisms, out-of-band or in-band, deduplicate inter-VM and intra-VM memory pages [10], [12], [13] or pages in the `tmem`-store [8], but not all of these at once. Synergy addresses this gap to provide a holistic hypervisor managed storage caching solution.

Architecture of the Synergy caching solution is shown in Figure 4. and it's components are, a *guest interface* to handle guest OS requests for disk objects, a *policy interface* to configure cache sizing, storage options and eviction policies, a *cache store* to maintain per-VM disk caches, and a *cache manager* to implement the backend for the two interfaces and manage the cache store. Synergy's cache store builds upon a KVM-based `tmem` caching implementation. The key idea to unify the memory management techniques has two components,

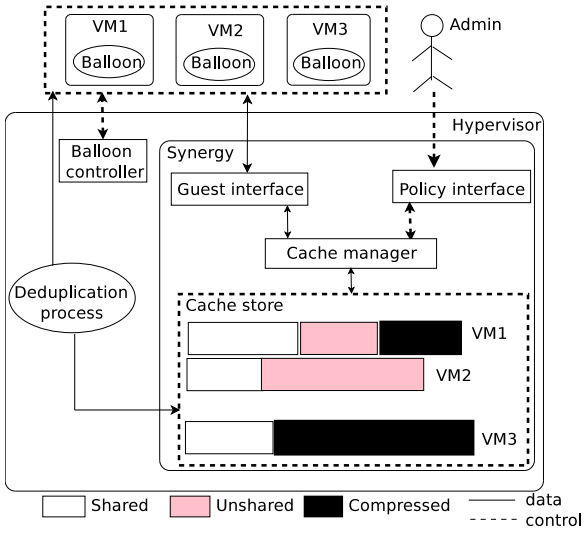


Fig. 4: Architecture of the Synergy caching solution.

(i) the deduplication domain of Synergy considers sharing opportunities within and across VMs and also includes the hypervisor resident cache.
(ii) the Synergy cache is not-only used as a second-chance cache, but also as a temporary store to *reshare* pages ballooned out from VMs to address the negative impact of broken sharing.

Synergy stores objects of two varieties—temporary pages that are expected to be reshared and pages within the second-chance cache that may or may not be shared based on content similarity and storage type. Further, pages stored for each VM depends on the configured storage type (Figure 4), e.g., VM1's state includes shared, unshared and compressed pages (elaborated later), while state of VM3 includes shared and compressed pages only. Additionally, Synergy provides a policy interface to implement cache sizing and access to statistics related to the usage and effectiveness of its cache to aid automated cache management.

3.3 Cache configurations

Efficient cache space utilization is a central concern for Synergy. Every VM can be individually configured to use any of the three storage options within the Synergy cache store as described below:

Shared-only (SHR): On a guest OS's `put` page request, the page is stored only if it can be shared with any other page in the system (through the deduplication process) at that instant of time. The `SHR` store consumes no additional memory space because it stores only blocks that already exist in the system elsewhere in the first place. Yet, it provides caching benefits when a `get` for the disk block is issued from a VM. In a *sharing friendly* setup, this caching mode can provide significant benefits in a memory efficient manner.

Shared+Unshared (SHU): On a guest OS's `put` page request, the page is added to Synergy cache irrespective of whether it can be shared or not. The page remains a candidate for sharing (if an opportunity arises in the future) until it is evicted from the cache.

Shared+Compressed (SHZ): On a guest OS's `put` page request, the page is compressed and stored if it cannot be shared with another page at the time of insertion. Otherwise, the page is

left uncompressed to be shared with another page in the system. While this mode enables increased memory utility, any sharing opportunity that may arise in the future can not be availed because out-of-band scanning deduplication mechanisms are unable to deduplicate compressed content.

4 SYNERGY IMPLEMENTATION

Synergy is implemented using the KVM hosted virtualization solution (Linux kernel version 3.2) by extending its `tmem`-based hypervisor caching solution. The guest OS file system interacts with the Synergy caching system using the `cleancache` interface [19] available as part of the Linux kernel. This interface is similar to that of Xen's `tmem` and KVM's `zcache` implementations [8], [17]. Synergy's interface to guest OSes can be implemented through a hypercall interface [20] or using the virtio interface [20], [21]. In Synergy, a shim layer in the guest OS translates `cleancache` calls into hypercalls to the KVM hypervisor. These hypercalls are handled by the Synergy interface inside the hypervisor and appropriate actions initiated on the cache store.

4.1 System-wide deduplication

To achieve system-wide exclusive disk caching, Synergy uses the Kernel Same-page Merging (KSM) daemon [10], an out-of-band scanning based content deduplication service in the Linux kernel. A process wanting to share pages, offers its virtual address space to KSM for scanning and merging using the `madvise` [22] system call. With KVM, the VM address space is a process's address space from the host context, and is made sharable by QEMU [23] (which is part of the KVM solution).

The KSM scanner thread of the KVM host kernel scans through the virtual address space of all VMs (processes) to *compare and merge* content-identical physical pages. KSM implements page merging across the VMs by modifying the respective page table entry of each VM process to point to a single physical page. The freed-up replica pages are now available for use by the KVM host kernel. The shared page is marked *read-only* by updating the appropriate page table entries; any write attempt causes a page fault which is handled using the *copy-on-write* (CoW) mechanism.

As a result of KSM's design, memory pages that can be shared by KSM *must* be accessed through a hardware page table walk. Any update to the state of a shared page that bypasses the MMU would need to be carefully coordinated with KSM management of the page. For instance, page evictions and DMA to kernel pages by the hypervisor page cache manager will not cause a page fault as access to page is not through the hardware MMU. These may result in inconsistent page content.

KSM's dependence on CoW-based protection to correctly handle updates to shared pages is the primary reason that its mechanism is limited to process *anonymous pages*. One could address this limitation by changing the KSM design to additionally handle hypervisor-managed (e.g., `tmem`) pages without any process mappings. This approach is intrusive and requires significant changes to KSM and the Linux kernel.

We implement an alternative solution that is simple and far less intrusive. For each VM, we create a dummy `promoter` process that allocates a range of virtual addresses without any physical memory allocation. VM pages cached within Synergy are mapped to virtual addresses of the dummy process to *promote* them to be part of the KSM deduplication process. The promoter process' address space is marked as *sharable* to KSM. Per-VM `promoter` processes enable parallel execution of `tmem` operations from multiple VMs.

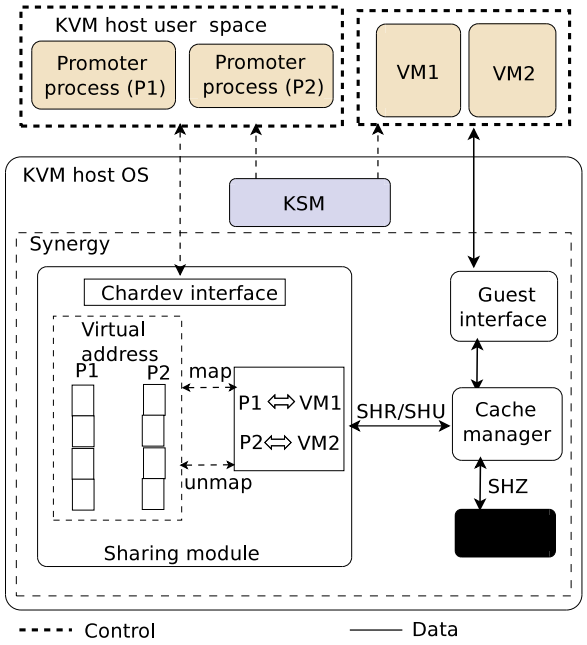


Fig. 5: Integration of Synergy caching system with KSM service of KVM host.

Even without system-wide deduplication, vanilla `tmem` and KSM *do not integrate* in a seamless manner because of KSM's reliance on the CoW semantics. Consider the case of a `get` operation where the guest OS provides a shared guest page frame number (gPFN) to retrieve content from the `tmem` cache. In the host kernel, the guest's gPFN is converted to machine frame number (MFN), the MFN is mapped to a temporary host kernel virtual address, and the memory page containing the disk page is copied to the MFN page. This leads to bypassing of KSM's CoW semantics and inconsistent page content for some other VM whose physical frame (gPFN) points to the shared MFN page. To overcome this problem in Synergy, the guest shared machine page (gPFN) is explicitly CoW broken to create a separate copy. Further, the corresponding new guest virtual address is temporarily made *non-sharable* by KSM until the page cache content is copied into the guest.

4.1.1 The Sharing module

The Synergy sharing module (shown in Figure 5) acts as a glue between KSM and the Synergy system. Synergy provides hooks that are implemented by the sharing module to provide the following functionality: *(i)* map and unmap disk pages that correspond to `put` and `get` operations, *(ii)* VM register and deregister events. The sharing module also implements a character device interface for the user space `promoter` process to communicate with the module.

During system initialization, each `promoter` process allocates a range of virtual addresses and makes these available to KSM for sharing using the `madvise` system call. Further, the virtual address range is communicated to the sharing module via the device interface. The `promoter` process remains asleep for rest of its lifetime. For security reasons, the character device and `promoter` process are accessible only in superuser mode.

An unused promoter process is assigned to a VM when the VM registers with the Synergy caching system. On a put operation by the guest OS, Synergy copies the content to a host page and *maps* it onto a free virtual address of the corresponding promoter process. If the page is sharable, KSM shares the page when the virtual address of the promoter is next scanned. For a get request, the virtual address of the promoter process is *unmapped* and the page content is copied onto the page provided by the guest OS.

4.1.2 Cache storage options

Implementation details of the storage options provided by Synergy for the cached objects are as follows,

Shared-only (SHR): When a guest OS requests to put a disk page, Synergy determines the gPFN to MFN mapping to check if the corresponding machine page is already shared by KSM. If the MFN is shared, the content is copied to a temporary page and the map interface of the sharing module is invoked. As the original page is shared, its copy will most likely be shared when KSM next scans the virtual address space of the promoter process. Therefore, the memory used to store the content is temporary and is of constant size. For a get operation, if the page is found in the Synergy cache, the page contents of corresponding virtual address of the promoter are copied onto the guest provided page. A flush operation removes mappings from the promoter process.

Shared+Unshared (SHU): This mode is implemented similar to the SHR-mode with the additional feature that a page is mapped to a promoter process virtual address even if it can not be shared during eviction from the VM. This mode has the additional advantage of providing improved sharing because all the pages added to the promoter process address space remain candidates for sharing in the future. However, the Synergy cache consumes additional memory depending on the sharing opportunities.

Shared+Compressed (SHZ): In this mode, non-sharable pages are compressed and stored using Linux's zcache[24], [17]. Other zcache features such as cache management and eviction etc. are not used by Synergy. This mode has the advantage of saving memory by intra-page deduplication for the cached pages and inter-page deduplication across VMs and the Synergy cache. However, the pages that are compressed are no longer candidates for sharing by KSM. Furthermore, synchronous compression and decompression can consume significant processor cycles [17]. Finally, the memory savings depend on the compressibility of used disk pages.

4.1.3 Optimizations to cache operations

Synergy modifies and augments implementation of tmem operations to reduce the sharing overheads. In tmem, a get request removes a disk page from the cache to maintain exclusivity, ensure disk page consistency, and to avoid overheads of cache write propagation from the VM to the tmem cache. Removal of shared pages from the Synergy cache on a get operation breaks the sharing and can lead to overheads due to increased number of CoW breaks. To avoid these CoW handling overheads, Synergy does not remove a shared disk page on a get request, but marks it unavailable for future get operations. The disk page becomes available when a put request for the same block either confirms validity of the cached block or updates the disk page with fresh content.

Algorithm 1 Procedure for cache partitioning across VMs.

procedure DOCACHEPART($MaxCacheSize, VMList[1..n], AllocTuple[1..n], UT[1..3]$)
▷ *MaxCacheSize*: Synergy system wide cache size limit,
▷ *VMList*: list of VMs, each with statistics like number of shared, compressed, total etc.
▷ *AllocTuple*: Allocation ratio tuple,
▷ *UT*: Utility estimation tuple
  $T_{aw} = 0$ ▷ Total allocation weight
  $T_{cu} = 0$ ▷ Total cache usefulness
  $T_{sf} = 0$ ▷ Total sharing friendliness
  for i=1 to n **do**
    vm_i = *VMList[i]*
    VMSTAT(vm_i, AllocTuple[i])
    T_{aw} += vm_i.*aw*
    T_{cu} += vm_i.*cu*
    T_{sf} += vm_i.*sf*
  end for
  $U = \sum_{j=1}^{3} UT[j]$
▷ Sum of all utility parameters
  for i=1 to n **do**
    vm_i = *VMList[i]*
    $w_i = \frac{UT[1]*vm_i.aw}{U*T_{aw}} + \frac{UT[2]*vm_i.cu}{U*T_{cu}} + \frac{UT[3]*vm_i.sf}{U*T_{sf}}$
    vm_i.*entitlement* = w_i * *MaxCacheSize*
    if (vm_i.*entitlement* $<$ vm_i.*size*) **then**
      vm_i.*eviction* = vm_i.*size* - vm_i.*entitlement*
    else
      vm_i.*eviction* = 0
    end if
  end for
  return $VMList$ ▷ VMList with eviction plan
end procedure

Algorithm 2 Procedure to estimate per-VM utility parameters.

procedure VMSTAT($vm, allocweight$)
▷ *vm*: virtual machine with statistics like #get, #shared etc.
▷ *allocweight* : allocation weight of the VM
  vm.aw = *allocweight* ▷ aw: allocation weight
  $vm.cu = \frac{vm.get}{vm.get + vm.flush}$ ▷ cu: cache usefulness
  $vm.sf = \frac{vm.shared}{vm.size}$ ▷ sf: sharing friendliness
end procedure

5 Cache Sizing and Eviction Policies

Synergy provides the following cache configuration options: *(i)* per-VM cache limits, *(ii)* total Synergy cache size limit, and *(iii)* low system memory threshold (used for pro-active eviction of cached pages). Synergy's cache size and per-VM limits are specified in terms of number of cached disk pages, i.e., *inclusive of shared, compressed and unshared disk pages*.

5.1 Dynamic cache sizing

Cache sizing to meet application QoS (e.g., read latency) or for fairness of cache resource usage have been proposed [25], [26]. To facilitate the design of policy driven dynamic cache partitioning across VMs, Synergy provides two control knobs—the *allocation ratio tuple* and the *utility estimation tuple*. Outline of the cache partitioning scheme is shown in Algorithm 1. The Synergy maximum cache size limit (*MaxCacheSize*), an allocation tuple

and an utility estimation tuple (explained below) are used as inputs to decide cache sizing for each VM. The utility estimation tuple (referred to as *UT*) is used to estimate cache utility of each VM and it comprises of three configuration parameters— (i) weightage to allocation ratios, (ii) *cache usefulness*, ratio of read requests to total (read+write) Synergy cache accesses, and (iii) *sharing friendliness*, ratio of shared objects to total cached objects. The allocation ratio tuple (*AllocTuple*) specifies the relative Synergy cache size of each VM and is applicable only if the weight-to-allocation-ratios knob is a non-zero value in the utility estimation tuple (*UT*). For example with four VMs, if the allocation ratio tuple is set to <1, 1, 1, 1> and the utility estimation tuple is set to <1, 0, 0>, the cache and sharing friendliness parameters are not considered and the cache is partitioned equally among the four VMs. Alternatively, for the same setup, if the utility estimation parameters are set to <1, 1, 1>, the allocation ratio and the friendliness parameters are used with equal "importance" to determine the utility of the cache for each VM, which in turn is used as a ratio to partition the cache. Different cache partitioning policies can be instantiated by configuring the two tuples appropriately.

Per-VM statistics—number of `get`, `put`, `flush` requests, are maintained by Synergy. Further, the Synergy cache size of each VM (*vm.size*) and its break up in terms of shared, compressed or uncompressed pages are also maintained (referred to as *vm.shared* etc. in Algorithm 2). Utility of each VM w.r.t. priority, cache friendliness and sharing friendliness are calculated using the `VMSTAT` procedure (Algorithm 2). In Algorithm 1 (line# 15), comparative importance of utility estimation tuple is considered to calculate the weighted score for the VMs. Finally, the weighted score is used to calculate per-VM Synergy cache entitlement and evictions (if required). For brevity, the procedure `DOCACHEPART` assumes non-zero values for T_{aw}, T_{cu} and T_{sf}. An example of cache partitioning using Algorithm 1 for two VMs with following properties is shown here. vm1{*get=1000, flush=500, shared=100, size=200*}, and, vm2{*get=800, flush=700, shared=50, size=200*}. With allocation ratio tuple <1, 1> and utility estimation tuple <1, 1, 1>, vm1.aw, vm1.cu and vm1.sf are calculated to be 1, 0.66 and 0.5, respectively. Similarly, vm2.aw, vm2.cu and vm2.sf are assigned values 1, 0.53 and 0.25, respectively. Next, T_{aw}, T_{cu} and T_{sf} are calculated to be 2, 1.19 and 0.75, respectively. Finally, using the above values, w_1 and w_2 become 0.573 and 0.427, respectively. If the Synergy cache size limit is 1000MB, cache partition for the two VMs are 573MB and 427MB, respectively.

5.2 Read-ahead aware eviction policy

In Linux, file system read-ahead is tightly integrated with the page cache [27]. The read-ahead logic in the guest OS decides to prefetch a chunk of file pages including the requested file page. For example, a disk page (corresponding to say block # 1000) is not found in the page cache and the read-ahead logic of the file system (in the guest OS) decides to prefetch a chunk of the file (say blocks 1000 to 1010). When the guest OS performs a sequential read-ahead, a Synergy cache lookup (`get`) is performed for all the disk pages in the read-ahead chunk. The utility of the Synergy cache is high if all the read-ahead pages are present in the cache. On the contrary, partial hits in the Synergy cache still require disk reads, thereby nullifying most benefits from partial hits. In the above example, if page corresponding to block# 1005 is not in the Synergy cache, a corresponding cache miss (failed `get` request) will have to serviced by the file system in the guest OS. Our proposed eviction method attempts to minimize the partial read-ahead hits, which we refer to as *read-ahead fragmentation*. To realize this, Synergy applies the eviction at a file object level instead of the page granularity. All pages from the victim file or continuous pages from the victim file are evicted from the Synergy cache depending on the required eviction size. This scheme minimizes the read-ahead fragmentation for sequential workloads. To handle random file operations, a generic method could classify file objects into two groups—sequential and random, to apply a combination of file and page level eviction policies. We leave this as a future direction of work.

Algorithm 3 Eviction of file blocks from a VM

```
1:  procedure EVICTBLOCKS(vm, EvictionSize)
    ▷ EvictionSize: Total number of cached blocks to be evicted
    ▷ vm: The victim virtual machine
2:      SortedObjList = sort_fileobjs_by_utility(vm)
3:      count = 1
4:      while EvictionSize >= 0 do
5:          FileObj = SortedObjList[count]
6:          if EvictionSize >= FileObj.count then
7:              EvictionSize -= FileObj.count
8:              synergy_evict_object(FileObj)
9:          else
10:             synergy_evict_continuous(FileObj, EvictionSize)
11:             EvictionSize = 0
12:         end if
13:         count++
14:     end while
15: end procedure
```

Importance aware victim selection for caches is a well researched area. Different cache replacement algorithms like LRU, ARC [28], CAR [29], MQ [30] etc. use access frequency and/or access recency as parameters for deciding the importance of a cache page. To decide the victim file object in Synergy, we consider two additional aspects along with access recency—sharability of the file object, and the reads to writes ratio for the file object. We have introduced accounting information into each file object—*last access time*, *storage type counts* like shared, compressed and plain text, and number of *read, write* counts. *Utility* of each file object is calculated as a function of—(i) last access time, (ii) read to write ratio, and (iii) shared blocks to total blocks ratio. A simple weighted function, used by Synergy to estimate utility of a file object f is given below,

$$u_f \quad = \quad 100 \times \left(\frac{s_b}{t_b} + \frac{g_b}{g_b + f_b} \right) \tag{1}$$

where, s_b is the number of shared blocks, t_s the total blocks of file in the cache, g_b is the number of `get` requests and f_b the number of flushed blocks. An additional fifty utility points are added for file objects that are accessed recently (in last 5 seconds).

An outline of the algorithm to evict disk blocks from a VM is shown in Algorithm 3. File objects are sorted in increasing order of utility (Equation 1); evictions continue from lowest utility to highest utility till requested number of Synergy cache blocks are evicted. All blocks of a selected file object are evicted (`synergy_evict_object`) if the eviction request is more than the number of cached blocks (*FileObj.count*) for the file object. If number of cached blocks for file is less than the eviction request,

Workload	Description	Nature	Sharing
ccode	kernel source browsing	Read intensive	Low
webserver	filebench [31] workload	Read intensive	High
twitter [32]	Micro-blogging workload	Read intensive	Low
TPC-C [32]	OLTP benchmark	Write intensive	Low
fileserver	filebench workload	Write intensive	High

TABLE 4: Workload details for experimental evaluation.

a specialized interface (synergy_evict_continuous) is invoked to evict continuous blocks from the file object to *minimize read-ahead fragmentations*. Note that, the VMs accessing Synergy cache through get, put and flush operations use file inode number and block offset within the file as part of the key.

6 Experimental Evaluation

Evaluation of Synergy was performed with two main objectives. First, characterize the effectiveness of Synergy enabled features—system-wide deduplication, ballooning-sharing co-existence and read-ahead-aware eviction policies (§6.1 to §6.3). These experiments were performed on a Intel Quad core i5 machine with a 2.8 GHz processor and 2GB RAM. Second, demonstrate the use cases of Synergy in a cloud hosting setup (§6.4)—*(i)* towards memory efficient VM placement, *(ii)* adaptive cache partitioning policies and *(iii)* to design holistic memory management policies. As workloads (refer to Table 4), we used two OLTP workloads—twitter and TPC-C , two filebench [31] workloads—webserver and fileserver, and ccode, a synthetic workload which randomly selects files from the kernel source tree and reads the whole file sequentially.

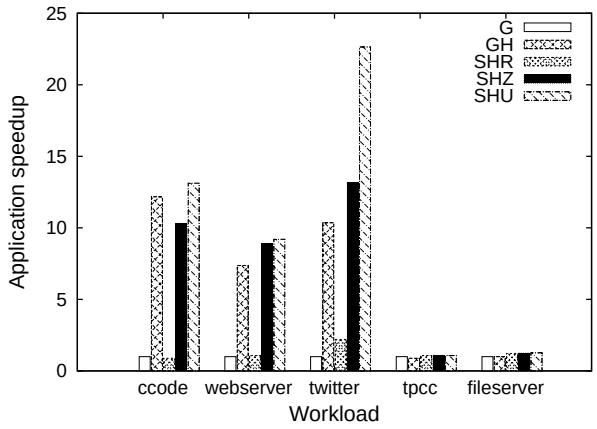
With a hosted hypervisor like KVM, the VM can be configured in one of many possible disk cache configurations to expedite disk I/O. The disk cache configurations used for experimentation were, page caching only by the guest OS (denoted as G), page caching in guest and the host (the default KVM caching mode, denoted as GH) and the different caching modes of Synergy (SHR, SHU and SHZ). Except for the GH mode, the host's native page cache was *disabled* for all other caching modes. For all our experiments, KSM was enabled with a scan rate of 8000 pages every 200ms [6].

6.1 Comparison of Synergy caching modes

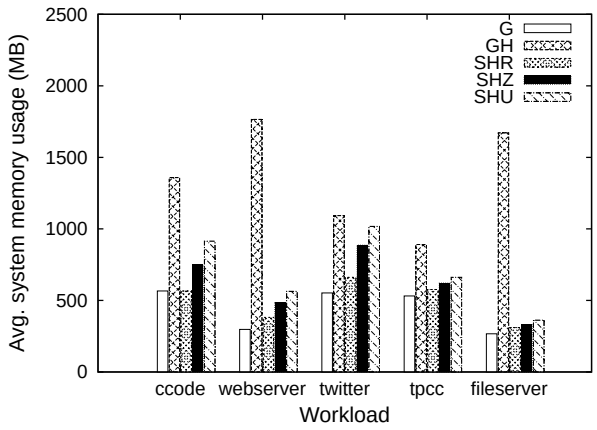
To evaluate different caching modes, workloads were executed within a single VM (512 MB RAM, 1 VCPU) with no limits on the Synergy cache size.

6.1.1 Application performance improvements

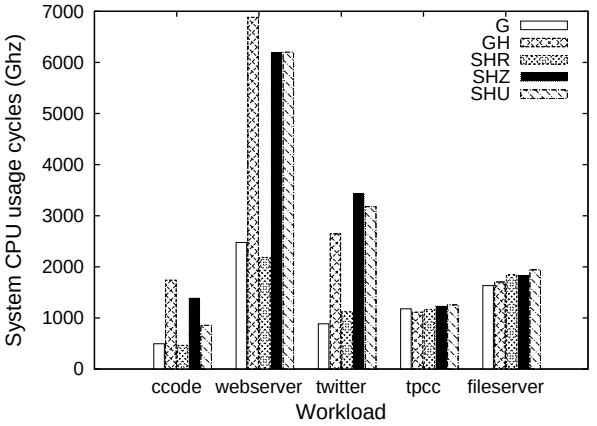
The application-level benefits for different workloads are shown in Figure 6a. The Y-axis represents the application speedup—ratio of application throughput for a particular cache setting vis-a-vis application throughput with guest-only caching (G) . For the read intensive ccode, and webserver, and twitter workloads, GH, SHZ and SHU caching modes resulted in improved throughput, with application speedup for the SHU caching mode being the best (13x, 9x and 23x for ccode, webserver and twitter, respectively). The GH mode resulted in 12x, 7x and 10x throughput



(a) Application speedup and caching modes.



(b) Average system memory usage.



(c) Total CPU utilization.

Fig. 6: Comparison of hypervisor and Synergy caching modes.

improvements for ccode, webserver and twitter, respectively, but with significantly higher memory utilization (up to 3 times) compared to SHU (Figure 6b). For the write intensive workloads—fileserver and TPC-C, no noticeable application speed-up was observed due to either Synergy caching or host caching.

In this experiment, the SHR mode did not provide significant throughput improvements (Figure 6a) because of its dependency on sharing. For example, in case of the ccode workload, the average KSM shared memory was only 39MB, which resulted in a small number of pages added to the Synergy cache and resulted in a low Synergy cache hit ratio (around 1%). However, with the webserver workload, Synergy cache hit rate was

around 90% which resulted in a reduction of block layer I/O by ∼8x compared to G. Nevertheless, these improvements did not translate into application-level throughput benefits. We speculate that since sharing depends on KSM scan rates, by the time a page was `put` it may not have been shared and hence not added to the Synergy cache in `SHR` mode. Note that in `SHR` mode only shared pages are considered by the Synergy cache. Moreover, even when a page is shared and thus added to the cache, *read-ahead fragmentation* due to partial hits in the Synergy cache negate the cache hit benefit as a disk read becomes necessary to read the missing blocks. We observed $\sim 4 \times 10^5$ cases of read-ahead fragmentation for `SHR`. An interesting take-away for the design of hypervisor managed caches is that *a reduction in the number of disk reads does not necessarily improve application performance*.

6.1.2 Memory and CPU efficiency

Figure 6b shows the memory utilization for each of the workloads and various caching modes. The `GH` caching mode used the highest memory because of inclusive caching of disk block contents and the host cache not being part of KSM sharing. For read intensive workloads, the Synergy caching modes, `SHU` and `SHZ` were memory efficient compared to `G` and `GH`. The memory efficiency (calculated as the throughput to memory utilization ratio) of `SHU` mode for the `webserver` workload was 4.5x and 3.9x compared to `G` and `GH`, respectively. Similarly, for the `ccode` workload, `SHZ` was 7.7x and 1.5x times more memory efficient as compared to the `G` and `GH` modes, respectively. For the `fileserver` and `TPC-C` workloads, approximately 100MB more memory was used in Synergy modes compared to `G`, without any application throughput improvements. In comparison, the `GH` mode used up more memory for write intensive workloads than the Synergy caching modes as shown in Figure 6b.

We also evaluated CPU overheads with Synergy and found that the CPU utilization increase was proportional to application benefits and compressibility characteristics of content (as shown in Figure 6c). For each workload, the ratio of application speedup w.r.t. increased CPU utilization was greater than 1, implying a positive trade-off. We found CPU overheads increased by a factor of up to 4x compared to `G`, while the application speed up increased by a factor of up to 20x. In cases where there was negligible application benefit, corresponding CPU overheads were also negligible (for `TPC-C` and `fileserver` workloads).

6.1.3 Utility of compressed Synergy cache

Next, we characterize the utility of Synergy `SHZ` mode compared to guest only (`G`) caching mode. A suitable comparison requires that the VM memory allocation accounts for the memory used by `SHZ` cache store. Consequently, we calculated the memory used by the Synergy `SHZ` cache and allocated it to the VM to perform this comparison (referred as $G_{\Delta Z}$). The $G_{\Delta Z}$ values for the `webserver`, `twitter` and `ccode` workloads are 712 MB, 887 MB and 724 MB, respectively. The workload settings were the same as described in the experiment setup in §6.1 and results are presented in Table 5.

For the `ccode`-workload, in comparison to G, $G_{\Delta Z}$ resulted in ∼1.3x performance improvement. However, in comparison to `SHZ`, $G_{\Delta Z}$ resulted in orders of magnitude lower application throughput (∼8x). For the `webserver`-workload, there was no visible improvement with the $G_{\Delta Z}$ mode over the guest only caching mode (G). However, for `twitter`-workload, $G_{\Delta Z}$ out performed Synergy `SHZ` caching mode (and also `GH`) because the dataset for this workload fits in the guest page cache (∼99% guest page cache hit ratio). Further, this is attributed to the additional access and compression overheads of the second chance cache.

Cache settings	ccode Throughput (MB/sec)	webserver Throughput (MB/sec)	twitter Throughput (Req/sec)
G	6.24	20.1	44.2
$G_{\Delta Z}$	8	20.0	1598
SHZ	64.38	178	581

TABLE 5: Performance comparison of VM provisioned with additional memory used by `SHZ` storage mode.

VM configuration	Cache settings	Over-committed memory			
		1 GB	3 GB	5 GB	7 GB
C1	G	155.5	24.7	17.7	24.6
8 GB, 4 vCPUs	GH	28.5	15.05	14.1	17.3
	SHU	147.5	43.2	42.5	44.2
WSS: 6 GB	SHZ	150.2	45.3	42.8	76.8
C2	G	102.3	17.1	12.9	11.4
16 GB, 4 vCPUs	GH	15.2	8.9	8.8	8
	SHU	101.3	28	27.8	22.9
WSS: 12 GB	SHZ	100.7	27.7	27.5	23.4

TABLE 6: Comparison of application throughput per GB memory usage with varied over-commitment levels.

6.1.4 Effectiveness with over-commitment levels

To study effectiveness of Synergy with different levels of over-commitment, we performed the following experiment. Two configurations of VMs and workloads were used—C1, a VM with 8 GB memory and 4 vCPUs executed the `webserver`-workload with a ∼6 GB file working set, C2, a VM with 16 GB memory and 4 VCPUs executed the `webserver`-workload with a ∼12 GB file working set (Table 6). Application throughput per unit (1 GB) memory usage for the different cache configurations with different over-commitment levels is shown in Table 6. The over-commitment extent was generated using ballooning of corresponding sizes, e.g., with C1 an over-commitment of 3 GB corresponds to a reclamation of 3 GB using memory ballooning.

As expected, the first point to note is that with low levels of over-commitment all caching modes (except `GH`) provide the same effectiveness in both the configurations. This is primarily because working set size of the workload can be accommodated in the VM memory. As the level of over-commitment increases, `SHU` and `SHZ` outperform the `G` and `GH` modes. Over both the configurations, `SHU` and `SHZ` improved memory usage effectiveness by a factor of 1.6x to 4.5x compared to both `G` and `GH` (see Table 6). Note that throughput per GB memory was always better in case of `G` compared to `GH` (default hypervisor caching mode) because `GH` uses inclusive caching.

6.1.5 A summary

The key results of comparing Synergy's caching modes are as follows: *(i)* the memory effectiveness with inclusive caching (the `GH` mode) is poor relative to the Synergy caching modes, *(ii)* the `SHR` mode of Synergy, by itself does not provide significant benefits, but is beneficial when augmented with compressed or

	Throughput (MB/s)	Total pages	Shared pages	KSM shared (MB)
NoB	478	0	0	859
WithB	20	0	0	650
SHR	64	48777	48777	843
SHZ	178	48854	48203	845
SHU	187	48824	48509	880

TABLE 7: Performance of Synergy cache when combined with ballooning.

uncompressed storage, *(iii)* the memory effectiveness for write intensive applications is poor and needs to be addressed when sizing the Synergy cache, *(iv)* Synergy exploits system-wide sharing to improve the memory efficiency compared to other hypervisor exclusive caching solutions, and *(v)* Synergy is able to better utilize memory for sharing and increasing the throughput of read-intensive workloads, especially when the size of data to be read does not fit in the guest page cache.

6.2 Co-existence of ballooning and sharing

As discussed earlier, ballooning and sharing are orthogonal memory management techniques—shared pages may get ballooned out, break the CoW semantics, and reduce sharing potential. We designed an experiment to study the impact of the Synergy modes (`SHR`, `SHU` and `SHZ`) that enable co-existence of ballooning and system-wide sharing. The experiment used two VMs of size 512 MB each, executing the `webserver` workload, with five threads each. The workload was configured such that it inherently had a large potential for sharing and the I/O data set size could fit in a 512MB VM. 200MB of memory was ballooned out from one VM , while the other executed with the entire memory allocation of 512MB. The VM from which memory was ballooned out, was provided with a maximum Synergy cache size of 200 MB. For the other VM, the Synergy cache was disabled.

The performance of the VM whose memory was ballooned out is reported in Table 7. The baseline application throughput of the `webserver` workload without ballooning (`NoB`) and after ballooning (`WithB`) was 478 MB/sec and 20 MB/sec, respectively. The `SHR`, `SHU` and `SHZ` results in Table 7 report the findings for the VM for which 200MB of memory was ballooned out. The cache in the `SHZ` and `SHU` modes used up a maximum of 50 MB of the 200 MB reserved for Synergy. This additional memory was used to service 48K pages (through the `put` requests) and more than 98% of these pages were reshared. The number of pages shared by KSM without ballooning and with Synergy was similar (850 MB worth of pages). The application throughput of the `SHU` and `SHZ` modes was 178 MB/sec and 187 MB/sec, respectively. As can be observed, Synergy is able to reshare and maintain the levels of sharing even when shared pages are broken as part of the ballooning process. Further, application performance benefits of this sharing are high (up to 8x compared to the default ballooning setup). The performance gap with respect to th no ballooning case is attributed to the overheads of interaction and the operations of the second chance cache. System memory usage (with 2VMs) with ballooning and different Synergy cache configurations is shown in Figure 7. Extra memory used by the Synergy caching modes is marginal compared to ballooning with no Synergy cache setup (marked as `WithB` in the Figure). This result signifies the benefits

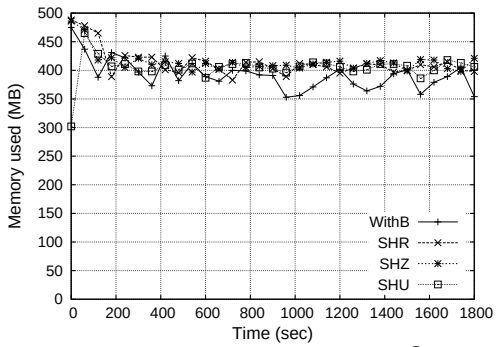


Fig. 7: System memory usage with different Synergy caching modes and ballooning.

of resharing evicted pages due to guest OS memory pressure (through ballooning) to satisfy future `get` requests.

6.3 Efficacy of Synergy eviction policies

To test the implications of the proposed read-ahead aware *file-level* eviction policy, we compared its effectiveness to the eviction policy of `zcache`, the KVM implementation of compressed `tmem` cache store. `zcache` uses a block-level LRU based eviction policy. To make the comparison fair, we used the `SHZ` caching mode with system-wide sharing *disabled*. A single VM configured with 512 MB of memory executed the `webserver` workload with varying number of threads. The Synergy cache size for the VM was configured to be 400 MB.

Application layer throughput for different numbers of `webserver` threads is shown in Table 8. In all cases, Synergy's file-level eviction policy resulted in better application throughput—2x to 3.5x improvement. Improved application throughput can be explained by examining Synergy cache hits which was higher than that of the `tmem` cache for all the cases. With 8 or more I/O threads, throughput for both solutions reduced significantly due to increased contention at the hypervisor cache.

We also instrumented the disk read path in the guest kernel to determine the number of *disk read-ahead fragmentations* based on the hypervisor cache hits and misses. As shown in Table 8, the number of occurrences of read-ahead fragmentation were significantly higher in the case of `tmem` and negligible in case of Synergy. The experiment demonstrates suitability of evicting files or continuous blocks of file by a hypervisor managed second chance cache to avoid read-ahead fragmentation.

6.4 Synergy Use Cases

Synergy provides improved capabilities for efficient memory management in over-commitment based cloud hosting solutions. In this section, we demonstrate three use cases for the same. The experiments presented in this section used a machine equipped with an Intel Xeon E5507 processor and 24GB RAM was used. VMs were configured with 1GB RAM and 1 VCPU.

6.4.1 Memory efficient VM hosting

To demonstrate the cost-effectiveness of Synergy in terms of memory usage we experimented with placement of homogeneous and heterogeneous VMs. The memory usage of inclusive caching (`GH`) is inefficient as shown in §(6.1.2). Therefore, we have considered two exclusive caching solutions—Singleton [14] and compressed `tmem` cache (a.k.a. `zcache`) [17], for the comparative analysis. Singleton periodically scans the hypervisor page

# IO threads	Block level LRU based eviction (tmem)				Synergy file level eviction			
	Throughput (MB/sec)	#read-ahead fragmentations	#get requests ($\times 10^5$)	#get hits ($\times 10^5$)	Throughput (MB/sec)	#read-ahead fragmentations	#get requests ($\times 10^5$)	#get hits ($\times 10^5$)
1	45.5	64633	213	153	93.8	138	439	418
2	46.2	111780	216	156	114	329	534	499
4	43.2	154768	202	140	119.7	237	560	514
8	37.4	205613	175	112	102.1	320	478	422
16	22.7	245086	106	37	73.9	398	346	280
32	19.7	11796	92	0.4	47.2	149	221	148

TABLE 8: Comparison of block vs. file level evictions in the hypervisor cache.

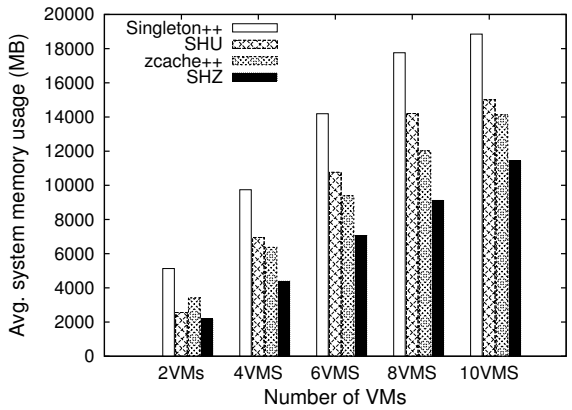


Fig. 8: Memory efficiency of Synergy compared to other exclusive caching solutions.

Singleton++	SHU	zcache++	SHZ
7464 MB	5844 MB	5808 MB	5203 MB

TABLE 9: Average memory utilization for heterogeneous workload mix.

cache to evict pages corresponding to disk blocks already present in the guest page cache. We approximated Singleton (referred to as Singleton++) using the Synergy `SHU` caching mode, where the `SHU` cached memory was not deduplicated (only the VM memory was deduplicated using KSM). In Singleton++, hypervisor exclusive caching is achieved by explicit guest OS and Synergy cache `tmem` operations and does not require periodic scanning. `zcache` offers a compressed `tmem` store implementation. The `SHZ` caching mode without system-wide deduplication, but with VM-level deduplication (referred as zcache++) is used for comparison.

For a `webserver`-workload with 2GB total IO size and 16 threads, system memory usage for varying number of VMs is shown in Figure 8. Synergy caching modes, `SHU` and `SHZ`, use memory more efficiently compared to Singleton++ and zcache++, respectively. For example, the memory used by Singleton++ for 6VMs is sufficient to provision 8VMs using the `SHU` caching mode. Similarly, up to 8 VMs can be provisioned using the `SHZ` mode in the same amount of memory used by zcache++ to provision 6 VMs. The CPU usage and per-VM application throughput of `SHU` and `SHZ` were similar to Singleton++ and zcache++, respectively.

To study the memory efficiency in a heterogeneous hosting scenario, we experimented with all the workloads (Table 4) executing on five different VMs. `SHU` and `SHZ` modes saved 1.4GB and 600MB of memory (Table 9) compared to Singleton++ and zcache++, respectively. The application performance in all cases was the same.

6.4.2 Adaptive cache partitioning

As discussed in §5, Synergy provides the allocation ratio tuple and the utility estimation tuple to determine the dynamic cache partitioning policy.

For this experiment, we considered three VMs executing the `webserver`, `fileserver` and `ccode` workloads. The goal was to formulate an efficient cache partitioning policy. We configured the maximum Synergy cache size to be 1200 MB. Different cache partitioning policies resulted in different application throughput (Table 10) due to changes in per-VM Synergy cache sizes. The initial policy used a fair distribution strategy—equal allocation ratios and the utility estimation tuple (setting A) ignored the utility aspects (i.e., cache and sharing friendliness). The Synergy cache was distributed equally across the three VMs as shown in Figure 9. Next, in order to improve throughput of the `fileserver` workload, the policy setting (Setting B in Table 10) changed the allocation ratio tuple to $<1, 2, 1>$. Due to the updated policy, the Synergy cache allocation size of the `fileserver` workload changed to 600 MB and the other two VMs were allocated 300 MB each (refer to Figure 9). However, there was no performance improvement for the `fileserver` workload, and as a matter of fact the throughput of the `webserver` and `ccode` workloads reduced due to reduction of their respective Synergy cache shares.

Taking cognizance of the lack of cache utility for write intensive applications an updated policy can be set to consider the cache friendliness of each VM (Setting C in Table 10). Based on this policy, as can be seen in Figure 9, the `fileserver` workload being write intensive, was given 200MB (half of the fair share) of Synergy cache, while the rest was equally shared by `webserver` and `ccode` (500 MB each). The effect of such partitioning was significant; the `fileserver` throughput did not change but throughput of the `webserver` and `ccode` workloads improved by 8% and more than 200%, respectively, compared to their fair allocation setting (i.e., Setting A in Table 10). Further, to take advantage of the sharing opportunities in the `webserver`-workload, sharing friendliness was considered in the utility estimation tuple (Setting D). The `webserver` workload was allocated the highest share, ∼550MB (Figure 9), which resulted in a throughput increase of 12% and 20% (Table 10) compared to settings C and A, respectively. Compared to setting C, the throughput of the `ccode` workload decreased by ∼30% because of ∼50 MB reduction in its cache share. However, compared to fair allocation (setting A), the `ccode` workload resulted in ∼50% throughput improvement.

6.4.3 System-wide memory management

Synergy features like system-wide sharing, ballooning-sharing co-existence and flexible second chance cache sizing can be used to design efficient *system-wide* memory management policies. We evaluated an *example policy* for three VMs (each configured with

Setting	Alloc. ratio tuple	Utility est. tuple	Application throughput MB/sec		
			Web	File	ccode
(A)	<1, 1, 1>	< 1, 0, 0>	71.5	27.2	55.4
(B)	<1, 2, 1>	<1, 0, 0>	68.5	27.8	45.0
(C)	<1, 1, 1>	<1, 2, 0>	76.5	27.7	125.9
(D)	<1, 1, 1>	<1, 4, 1>	85.5	27.5	82.1

TABLE 10: Performance with different cache partitioning policies.

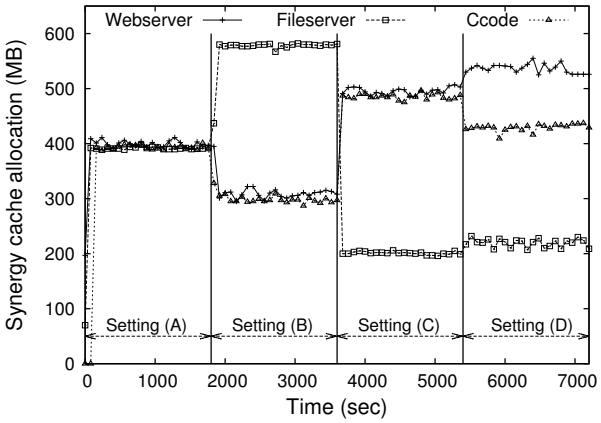


Fig. 9: Impact of partitioning policies on Synergy cache distribution.

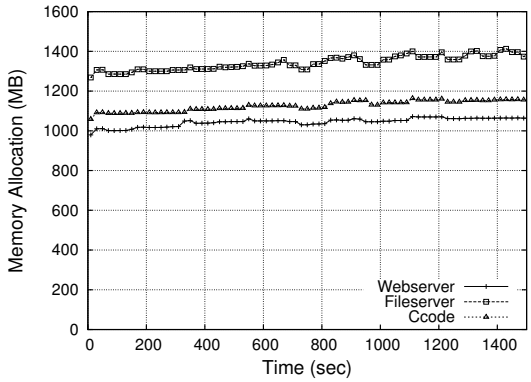
1.5GB memory) which executed the `webserver`, `fileserver` and `ccode` workloads. The aggregate memory usage limit was set to 3GB. The baseline balloon controller for our evaluation (referred to as CAS) adjusted the per-VM memory allocations in proportion to the `committed_AS` [33], [34] value reported by the guest OS. The Synergy controller (referred to as CAS-Syn) split the memory management into two parts— the CAS controller managed the anonymous memory requirements of the three VMs using ballooning-based memory adjustments, and the rest of the memory was managed by dynamic adjustment of the Synergy cache size. The cumulative anonymous memory requirements for the three VMs (for these workloads) never exceeded 1.5 GB and rest of the 1.5 GB was allocated and managed by the Synergy cache. The allocation ratio and the utility estimation tuple values were set to <1, 1, 1> and <1, 4, 1>, respectively.

Performance of the CAS controller and the CAS-Syn controller with the `SHU` and `SHZ` modes are shown in Table 11. Using the CAS-Syn (`SHU`) controller resulted in improved throughput— 2.8x and 1.9x for the `webserver` and `ccode` workloads, respectively, compared to the CAS controller. The improvements are attributed to system-wide deduplication resulting in 70% (Table 11) more memory savings compared to CAS, and efficient redistribution of the Synergy cache across the VMs. The memory savings were calculated by subtracting the memory limit (i.e., 3 GB) from the average allocated memory i.e., sum of VM allocations (for CAS) and sum of VM allocations + Synergy cache size (for CAS-Syn). With `SHZ`, significant memory savings (300% more compare to CAS) were recorded with application benefits similar to `SHU` mode.

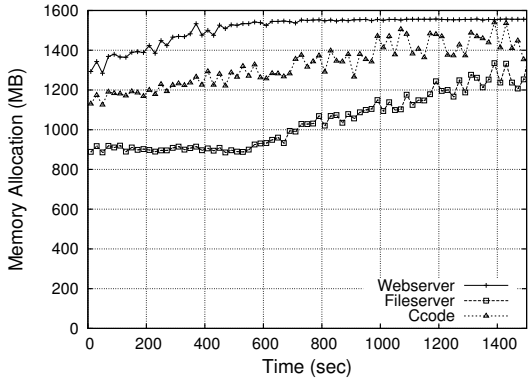
The steady state memory distribution across the three VMs with CAS, CAS-Syn with `SHU` caching mode and CAS-Syn with `SHZ` caching mode are shown in Figure 10a, Figure 10b and Figure 10c, respectively. CAS-Syn controllers redistributed the Synergy memory differently than the CAS controller. For example, the `fileserver`-workload was given comparatively less memory than the other workloads with CAS-Syn (Figure 10b) while with CAS controller (Figure 10a) it was the opposite case. The combined memory allocations for the three VMs was more

Policy	Application throughput (MB/sec)			Memory savings (MB)
	Webserver	Fileserver	ccode	
CAS	48	27.2	49.1	515
CAS-Syn (`SHU`)	133.4	25.5	85.72	881
CAS-Syn (`SHZ`)	135.5	30.2	76.8	1660

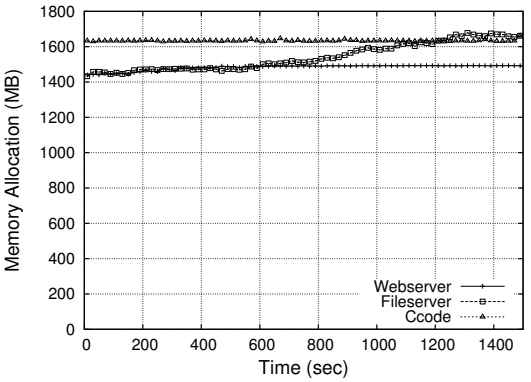
TABLE 11: Performance of an illustrative holistic memory management policy.



(a) CAS controller



(b) CAS-Syn controller with Synergy `SHU` mode



(c) CAS-Syn controller with Synergy `SHZ` mode

Fig. 10: Comparison of steady state memory allocation to VMs with different memory management controllers and Synergy configurations.

than 3GB for all the controllers as sharing (and compression) benefits keep the actual memory usage below 3GB. With CAS-Syn (SHU and SHZ) controllers, combined memory allocation to VMs was more than the CAS controller because Synergy could leverage system-wide deduplication benefits. For CAS-Syn with SHZ caching mode, the combined benefits of compression and sharing was sufficient to satisfy the memory requirement of the VMs in 3GB physical memory limit. Therefore, the cache allocations to the VMs were almost constant depending on the working set size of the VMs in this case (Figure 10c).

7 Related work

Hypervisor managed second chance caching for VMs is provided by different virtualization solutions [8], [9]. These implementations provide a key-value based interface and different caching semantics (ephemeral and persistent caches). Further, they also implement compression and sharing of objects within the hypervisor cache. In contrast, Synergy builds on these techniques and is a holistic disk cache management system for hosted virtual machines that effectively combines memory management techniques—*sharing* and *ballooning*. Further, Synergy demonstrates the inefficiency of block level cache eviction polices for hypervisor caches and proposes a file-level eviction policy to achieve a symbiotic co-existence with guest OS cache management policies.

Out-of-band page sharing techniques [10], [3] merge pages with same contents to reduce memory utilization of the system. Several enhancements (like XLH [18]) enable priority based scanning by tracking the nature of page usage. An alternate method, in-band deduplication on the disk access path [13], [35], shares memory pages used by the VMs for disk block caching. Difference engine [11] enables deduplication of memory at sub-page granularity. These techniques neither provide system-wide deduplication nor consider the impact of ballooning, both of which are addressed comprehensively by Synergy.

Dynamic memory re-sizing using ballooning [2], [3] enables a framework for dynamic memory management. However, accurate estimation of the memory needs, i.e., the working set size (WSS) of guest VMs remains challenging in virtualized systems. Several *black box* WSS estimation approaches have been proposed earlier [36], [3], [37], [38], [39]. Geiger[12] monitors guest OS disk accesses to build information regarding the guest page cache utility and uses this information for VM resizing. Gray box memory estimation methods intrude into the guest OS [33], [34], some even into the applications [40] to implement efficient balloon controllers. With respect to such balloon-based memory management approaches, Synergy complements them and increases memory efficiency by enabling the co-existence of ballooning and sharing techniques.

For hosted hypervisors such as KVM [41], the host cache can be a potential system wide second chance cache if the inclusive nature of caching can be addressed. *Singleton* [14] implements indirect mechanisms to remove disk blocks which are also present in the guest OS from the host OS page cache. For VMs that are created from the same template (CoW disks), Zhang *et al.* [42] developed a mechanism to store guest specific disk blocks inside the guest cache while the common content is stored in the host cache. However, neither of these works provide cache sizing flexibility nor were the host page caches considered as sharing candidates for system-wide deduplication.

Ex-tmem [43] extends the basic functionality of tmem to enable SSD storage as a third-chance cache for virtualized systems. Mortar [7] enables distributed caching applications (e.g., memcached[44]) to use a hypervisor managed cache (similar to tmem) to provide a distributed key-value store with elastic cache sizing. While both these approaches extend the basic tmem functionality, they are orthogonal to the goals of Synergy which is to provide system-wide deduplicated caching.

8 Conclusion

In this work, we highlighted the inefficiencies of combining the different memory management techniques—sharing, ballooning and hypervisor caching—and demonstrated the inability of these techniques to efficiently manage memory in over-committed setups. Towards addressing these shortcomings, we proposed Synergy, a holistic hypervisor caching solution. Synergy's unique design achieved a system-wide deduplicated disk cache with configuration knobs for cache sizing, selection of object storage types in the cache, and several caching policies. Synergy enabled the co-existence of the ballooning and sharing based memory management techniques to improve memory efficiency factors. An extensive experimental evaluation of Synergy implemented on the KVM platform demonstrated the cost-benefit tradeoffs made by the different Synergy caching modes and Synergy's capability to improve memory management in over-commitment based VM hosting platforms. Our evaluation showed that Synergy can provide 2x to 23x increase in IO performance while using 10% to 75% less memory compared to current hypervisor exclusive caching mechanisms. We also proposed and demonstrated extent of benefits of a novel file-level eviction policy for hypervisor caches. Towards efficient memory management in over-committed setups, we demonstrated Synergy benefits—tight packing of VMs, administrator controlled dynamic cache sizing and holistic memory management.

References

[1] VMware, "The Role of Memory in VMware ESX Server 3," http://www.vmware.com/pdf/esx3_memory.pdf.

[2] J. Schopp, K. Fraser, and M. Silbermann, "Resizing memory with balloons and hotplug," in *Proceedings of Linux Symposium*, 2006, pp. 313–319.

[3] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, Dec. 2002.

[4] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman, "An empirical study of memory sharing in virtual machines," in *Proceedings of the USENIX Annual Technical Conference*, 2012.

[5] C.-R. Chang, J.-J. Wu, and P. Liu, "An empirical study on memory sharing of virtual machines for server consolidation," in *Proceedings of 9th International Symposium on Parallel and Distributed Processing with Applications*, 2011, pp. 244–249.

[6] S. Rachamalla, D. Mishra, and P. Kulkarni, "Share-o-meter: An empirical analysis of ksm based memory sharing in virtualized systems," in *Proceeding of 20th International Conference on High Performance Computing (HiPC)*, 2013, pp. 59–68.

[7] J. Hwang, A. Uppal, T. Wood, and H. Huang, "Mortar: Filling the gaps in data center memory," in *Proceedings of the 10th international conference on Virtual Execution Environments*, 2014.

[8] D. Magenheimer, "Update on transcendent memory on xen," Xen Summit 2010.

[9] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, "Transcendent memory and linux." in *Proceedings of Linux Symposium*, 2009, pp. 191–200.

[10] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Proceedings of the Linux Symposium*, 2009, pp. 19–28.

[11] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: harnessing memory redundancy in virtual machines," *Communications of the ACM*, vol. 53, no. 10, pp. 85–93, 2010.

[12] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the buffer cache in a virtual machine environment," *SIGOPS Operating Systems Review*, vol. 40, no. 5, pp. 14–24, 2006.

[13] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: enlightened page sharing," in *Proceedings of the USENIX Annual Technical conference*, 2009.

[14] P. Sharma and P. Kulkarni, "Singleton: system-wide page deduplication in virtual environments," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, 2012, pp. 15–26.

[15] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," *SIGMETRICS Performance Evaluation Review*, vol. 23, no. 1, pp. 188–197, 1995.

[16] B. S. Gill and L. A. D. Bathen, "Amp: Adaptive multi-stream prefetching in a shared cache," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, 2007.

[17] D. Mishra and P. Kulkarni, "Comparative analysis of page cache provisioning in virtualized environments," in *Proceedings of 22nd conference on Modelling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2014, pp. 213–222.

[18] M. Konrad, F. Fabian, R. Marc, H. Marius, and B. Frank, "XLH: More effective memory deduplication scanners through cross-layer hints," in *Proceedings of the USENIX Annual Technical Conference*, 2013.

[19] "Kernel documentation: vm/cleancache," www.kernel.org/doc/Documentation/vm/cleancache.txt, accessed: 2015-09-18.

[20] 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," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, Oct. 2003.

[21] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Operating Systems Review.*, vol. 42, no. 5, pp. 95–103, Jul. 2008.

[22] M. Kerrisk, "Madvise(2) linux programmer's manual," www.man7.org/linux/man-pages/man2/madvise.2.html, accessed: 2015-09-18.

[23] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 41–46.

[24] J. Corbet, "zcache: a compressed page cache," www.lwn.net/Articles/397574/, accessed: 2015-09-18.

[25] R. Koller, A. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *Proceedings of the 12th International Conference on Autonomic Computing*, 2015.

[26] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015, pp. 174–181.

[27] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new linux readahead framework," *SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 75–84, 2008.

[28] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003, pp. 115–130.

[29] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004, pp. 187–200.

[30] Y. Zhou, J. F. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the USENIX Annual Technical Conference*, 2001, pp. 91–104.

[31] "Filebench," www.filebench.sourceforge.net/wiki/index.php/Main_Page, accessed: 2015-09-18.

[32] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, 2013.

[33] J.-H. Chiang, H.-L. Li, and T.-c. Chiueh, "Working set-based physical memory ballooning," in *Proceedings of the 10th International Conference on Autonomic Computing*, 2013.

[34] D. Magenheimer, "Memory overcommit ... without the commitment," www.oss.oracle.com/projects/tmem/dist/documentation/papers/overcommit.pdf, accessed: 2015-09-18.

[35] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," in *Proc. of the USENIX Conference on File and Storage Technologies*, February 2010.

[36] P. J. Denning, "The working set model for program behavior," *Communications of ACM*, vol. 11, no. 5, pp. 323–333, May 1968.

[37] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient mrc construction with shards," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 95–110.

[38] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li, "Low cost working set size tracking," in *Proceedings of the USENIX Annual Technical Conference*, 2011.

[39] W. Zhao and Z. Wang, "Dynamic memory balancing for virtual machines," in *Proceedings of the 5th International Conference on Virtual Execution Environments*, 2009, pp. 21–30.

[40] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 2013, pp. 337–350.

[41] A. Kivity, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, Jul. 2007, pp. 225–230.

[42] Z. Zhang, H. Chen, and H. Lei, "Small is big: Functionally partitioned file caching in virtualized environments," in *Proceedings of the 4th USENIX Conference on HotCloud*, 2012.

[43] V. Venkatesan, Q. Wei, and Y. Tay, "Ex-tmem: Extending transcendent memory with non-volatile memory for virtual machines," in *Proceedings of 16th conference on High Performance Computing and Communications*, 2014.

[44] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, 2004.

Debadatta Mishra Debadatta Mishra is a PhD student in the Department of Computer Science and Engineering at the Indian Institute of Technology, Bombay. His research interests include operating systems, virtualization and cloud computing, communication networks, and online network games. He received a masters in computer application from Utkal University, India.

Purushottam Kulkarni Purushottam Kulkarni received the B.E. degree in Computer Science from the University of Pune, India, in 1997 and the M.S. degree in Computer Science from the University of Minnesota, Duluth, in 2000. He received his Ph.D. in Computer Science from the University of Massachusetts, Amherst in 2006. Currently, he is an Associate Professor at the Department of Computer Science and Engineering, Indian Institute of Technology Bombay. His interests include operating systems, virtualization, cloud computing, computer networks and technology solutions for developing regions.

Raju Rangaswami Raju Rangaswami received a B.Tech. degree in Computer Science from the Indian Institute of Technology, Kharagpur, India. He obtained M.S. and Ph.D. degrees in Computer Science from the University of California at Santa Barbara where he was the recipient of the Deans Fellowship and the Dissertation Fellowship. Raju is currently an Associate Professor of Computer Science at Florida International University where he directs the Systems Research Laboratory. His research interests include operating systems, storage systems, persistent memory, virtualization, and security.