DoubleDecker: A cooperative memory management framework for derivative clouds

Anonymous Author(s)

Abstract

Derivative clouds, light weight application containers provisioned in virtual machines, are becoming viable and cost-effective options for infrastructure and software-based services. Ubiquitous dynamic memory management techniques in virtualized systems are centralized at the hypervisor and are ineffective in nested derivative cloud setups. In this paper, we highlight the challenges in management of memory resources in derivative cloud systems. Hypervisor caching, an enabler of centralized disk cache management, provides flexible memory or non-volatile memory management at the hypervisor to improve the resource usage efficiency and performance of applications. Existing hypervisor caching solutions have limited effectiveness in nested setups due to their nesting agnostic design, centralized management model and lack of holistic view of memory management. We propose DoubleDecker, a decentralized memory management framework, realized through guest OS and hypervisor cooperation, with support for efficient memory management in derivative clouds. The DoubleDecker hypervisor caching framework, an integral part of our proposed solution, provides interfaces for differentiated cache partitioning and management in nested setups and is equipped to handle both memory and SSD based caching stores. We demonstrate the flexibility of DoubleDecker to handle dynamic and changing memory provisioning requirements and its capability to simultaneously provision memory across multiple levels. Such multi-level configurations cannot be explored by centralized designs and are a key feature of DoubleDecker. Our experimentation with DoubleDecker showed that application performance can be consistently improved due to the extended set of memory provisioning configurations.

1 Introduction

A vital enabler for high consolidation ratios of virtual machines in virtualization based hosting platforms is memory overcommitment [38, 39]. The main idea of techniques relying in overcommitment exploit the tradeoff between dynamic memory provisioning and usage efficiency, and performance implications. The challenge of memory management in virtualized systems [2, 20, 39] and container-based systems [11, 22, 24] is the opaqueness between the resource manager (the hypervisor or the host operating system) and the resource user (guest operating systems or applications). Techniques like dynamic ballooning [32, 39], memory content deduplication [1, 39] and hypervisor caching [27, 28, 30] enable dynamic provisioning, increased memory efficiency and system level disk cache provisioning for efficient memory management in virtualized systems. While these techniques and associated policies have been studied extensively in virtualization setups [7, 30, 39, 43, 45], they have not been studied in *nested hosting* setups, or what we refer to as *derivative clouds*. A *derivative cloud* is a nested setup, virtual machines nested in virtual machines [41] or containers deployed in virtual machine [15, 34], the latter being the focus of this work. In the derivative model, an intermediate service provider builds her own service offering on top of service she rents, buys or leases. The services themselves could be smaller sized versions of the first-level service or could be applications and in general host other software-as-a-service or platform-as-a-service variants. From the resource management perspective, with container based derivative clouds, the container-based nested services are provisioned and managed from within a virtual machine and resources for virtual machines are managed by the hypervisor-level policy tools. The central proposition of this work is that resource management techniques employed with single-level virtualization setups are inadequate in the context of derivative clouds and require augmentation and re-design.

In this work, our focus is the use of hypervisor caching for effective disk cache memory management in derivative clouds. Utility of a disk cache depends on application access behavior, e.g., access patterns (sequential vs. random) and read-write ratios. Best effort disk caches employed by operating systems greedily consume all available free memory in the system (virtual machine in this case) to provision disk cache memory. Further, disk cache management decisions of a guest OS are localized to the virtual machine and are oblivious to memory usage and disk cache utility across the virtual machines hosted on the same physical machine. Hypervisor caching [26–28, 30, 33] provides a mechanism for system-wide management of disk cache memory in virtualized systems. Hypervisor caches can be setup and used in two different configurations. First, available free system memory can be used to setup a hypervisor managed second-chance cache. Size of the second-chance cache and per-VM partitioning can be implemented based on resource management policies. Second, dynamic adjustment of virtual machine memory allocations through techniques like ballooning [32, 39] to explicitly push the disk caching to the hypervisor, fully or partially. In fact, in the second configuration, the page cache of guest OSes can be completely offloaded to the hypervisor. The hypervisor in-turn can improve memory efficiency by employing per-VM differentiated provisioning, perform in-band compression and deduplication etc. [27, 30].

Conference'17, Washington, DC, USA
© 2017 ACM. 978-x-xxxx-xxxx-x/YY/MM...$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Applicability of hypervisor caching in a derivative cloud setup is hindered due to several reasons. Firstly, hypervisor cache management is agnostic to the execution entities within the virtual machine. For example, hypervisor caches cannot provide differential cache allocations to multiple applications executing within a virtual machine in a seamless manner. Enlightening the hypervisor cache with application level information as an alternative design is cumbersome and requires understanding and redesigning of the current integration framework. Secondly, in a derivative setup it may be useful for guest OSes to holistically manage all memory resources—allocated virtual machine memory and the hypervisor cache. The current hypervisor caching solutions lack the capability to expose interfaces to provide visibility and local provisioning capability to the guest OS.

A desired memory management model in derivative clouds is to provide *independent memory management* at each level, without non-deterministic side effects and without loss of hypervisor control. The provisioning policies at each level should be mutually exclusive and at the same time allow each hosting entity to influence the memory provisioning decisions. For example, a hypervisor level policy can be used to implement VM level priorities and application level priorities may be configured within each VM for differential treatment for provisioning the VM-specific hypervisor cache. This design allows a VM-level memory manager to enforce comprehensive policies beyond the hypervisor cache provisioning. For example, a VM-level policy has the flexibility to meet application performance by provisioning in-VM memory and the hypervisor cache resources. We design techniques to address the above requirements to provide nested container level differentiation for hypervisor cache management with nested policy enforcement capability.

Further, recent advancements in memory technologies enable cost-effective storage mechanisms like non-volatile memory (NVM) and solid state devices (SSD). One of the compelling usages of NVMs is to employ them as second chance caches or hypervisor caches in the disk access path to improve disk access performance [35, 37]. Provisioning of NVM-backed hypervisor caches in a derivative setup suffers from the same limitations as that of memory backed hypervisor caches. A hypervisor caching solution that implements nesting aware policies (and by implication application level policies) irrespective of the storage methods is desirable.

Towards providing decentralized and differentiated memory management in a derivative cloud setup, we make the following contributions,

- We design and implement DoubleDecker, a hypervisor caching framework with differentiated partitioning capabilities across application containers along with VM-level provisioning capabilities. Our implementation of DoubleDecker is with the KVM virtualization solution and the LXC container framework [24].
- We demonstrate the features and efficacy of DoubleDecker to support nested memory management. Specifically, its flexibility in terms of dynamic and elastic provisioning capabilities and configurable cache storage options.

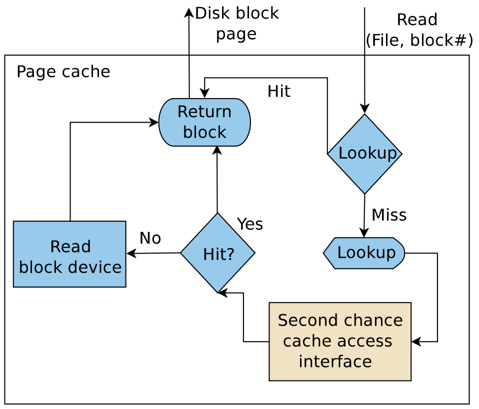


Figure 1. Cache lookup of file blocks in the page cache-second chance cache integrated caching systems.

- We show the flexibility of DoubleDecker to adapt to changing demands as the basis for designing memory management policies in a derivative cloud setup through efficient provisioning of disk cache resources.

2 Background and Motivation

With the Linux+KVM virtualization solution page cache of the host operating system acts as an inclusive cache. Previous studies [30, 33] have shown that inclusive caching can be wasteful due to duplication of disk blocks across the VMs and the host. In this paper, we focus on exclusive cache designs [28] like transcendent memory [28] where a symbiosis is proposed between the page caches of virtual machines and the hypervisor managed second chance cache.

2.1 Exclusive second chance caching

With an exclusive second chance cache, application initiated file IO requests are looked up in the OS managed disk page cache. On a page cache miss, requests are queried (lookup operation) in the second-chance cache (Figure 1). The second chance cache interface is tightly integrated with the page cache layer and extends the IO path. When a clean disk block is evicted from page cache, a store operation is initiated through the second chance cache interface. To maintain exclusivity between the two caches, a block evicted from the page cache is stored into the second chance cache and a block read from the second chance cache is removed from the second chance cache and transferred to the page cache.

A generic second chance cache interface is implemented through a cache store implementation using a storage option (e.g., memory or SSD), storage optimizations (e.g., compression, deduplication) and, cache management policies for eviction and cache partitioning. Linux cleancache [23], an abstract second chance cache access interface, provides a generic second chance cache integration framework with the page cache. The key operations of the interface are:—lookup (get) a disk block, store (put) a block in the cache and flush to invalidate objects from the cache (refer to Figure 2). In a virtualized setup, where second chance cache is implemented in the hypervisor, above operations ensure exclusive caching

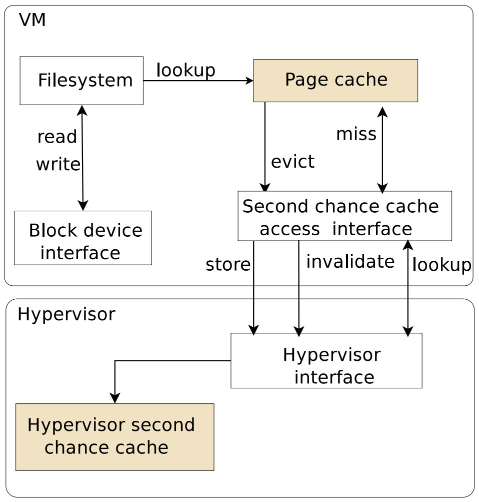


Figure 2. Overview of second chance cache integration with OS page cache with exclusive mode of operation.

between the guest disk cache and the hypervisor cache. Hypervisor managed caches interface with the guest OS second chance cache interface using the hypercall (VMCALL) API (Figure 2). The second chance stores only clean pages evicted from the page cache and is indexed based on file system mount points or virtual machine identifiers, the file information and the block number. Several second chance cache implementations for native Linux and for different virtualization solutions exist [9, 18, 27, 30]. However, no state-of-the-art solution supports application cognizant second chance cache management, especially for derivative cloud setups.

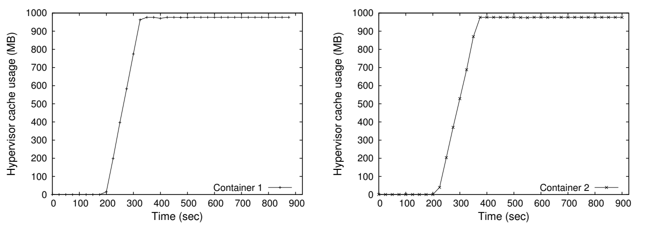
2.2 Application container isolation framework

Light-weight resource isolation frameworks like Linux control groups (Cgroup) [22] provide performance isolation across multiple execution domains with low virtualization and multiplexing overheads compared to hypervisor enabled system virtualization solutions. Linux Cgroup [22] and FreeBSD Jails [16] are examples of performance isolation frameworks facilitating resource management for a group of processes by specifying group-level resource limits. Using the process grouping based isolation model, container management solutions like LXC [24] and Docker [11] enable hosting multiple applications in different application groups (containers).

With containers in virtual machines derivative clouds, the two resource controllers—the hypervisor and the Cgroup subsystem—operate in an agnostic manner w.r.t. each other. As a result, centralized/global hypervisor cache distribution from the containers perspective is not deterministic (§2.3). To enable differentiated partitioning of the hypervisor cache and to facilitate intelligent use of multiple storage backends across containers, it is necessary to design a solution that can provide synergy across the two control points.

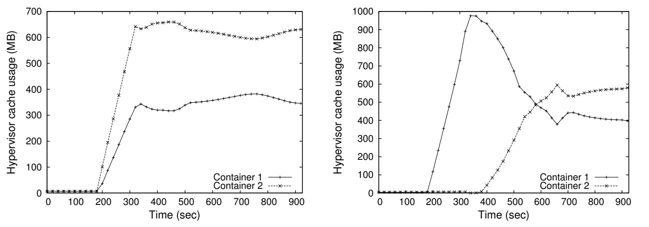
2.3 Motivation

To demonstrate the non-deterministic cache distribution in a nested container setup, we performed an experiment with a VM configured with 2GB memory and 4 VCPUs on a host



(a) Application container 1 **(b)** Application container 2

Figure 3. Hypervisor cache distribution when application containers executed separately.



(a) Same start time **(b)** Container 2 offset by 200 sec

Figure 4. Hypervisor cache distribution with different start times of application containers.

with 32GB memory and 16 CPUs. The hypervisor cache was not cognizant of the containers and its size was limited to 1GB. Two application containers (Container 1 and Container 2) configured with the same memory limits (through Cgroup) executed the Webserver-workload of the Filebench workload suite [36]. The only difference between the two containers was their IO load; Container 1 executed two Webserver threads while Container 2 executed three Webserver-threads.

With this setup, when Container 1 and Container 2 executed separately one-at-a-time, each of them used up the entire hypervisor cache (Figure 3). The implication being that each application is capable of taking advantage of the configured hypervisor cache to its capacity when executed separately.

When both applications were started simultaneously, the hypervisor cache was distributed across containers in a disproportionate manner. Share of Container 2 was approximately two times that of Container 1 as shown in Figure 4a. The non-determinism is a side-effect of the IO load and the FIFO-based global eviction policy (not ensuring container level fairness) of the hypervisor cache. When Container 2 started executing its workload after a delay of 200 seconds form start of workload in Container 1, Container 1 dominated the cache usage till about 500 seconds, after which cache share of Container 2 increased before surpassing that of Container 1 at around 600 seconds(Figure 4b). These results demonstrate that state-of-the-art hypervisor caches are unable to implement policies (e.g., fair allocation to application containers) in a deterministic manner at a sub-VM granularity.

2.3.1 VM level memory management flexibility

To analyze the impact of memory provisioning at the two levels (guest OS and hypervisor cache) in a nested setup

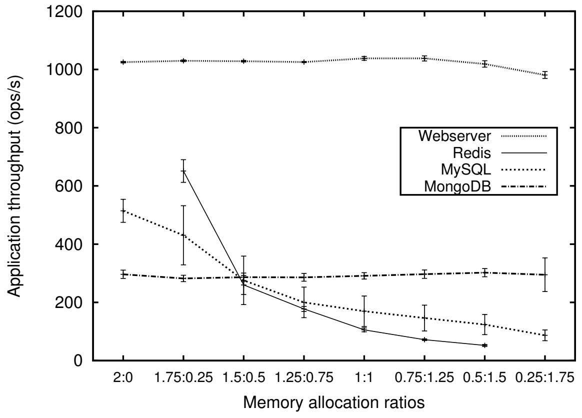


Figure 5. Application performance with different provisioning configurations at the two levels of a derivative cloud.

Application usage	Total swap (MB)	Anonymous memory usage (MB)	Hypervisor cache usage (MB)
Websever	0	88.9	1023
Redis	996	1021	18.5
MySQL	879	1013	34
MongoDB	0	251	1023

Table 1. Guest OS metrics for different applications with equal split (1GB in-VM and 1GB hypervisor cache) scenario.

and motivate the need for guest OS control on the hypervisor cache sizing, we performed the following experiment. 2GB memory was split in different ratios to allocate memory for the container inside the VM (through Cgroups) and the hypervisor cache. For example, an allocation ratio of 1.5:0.5 (Figure 5) refers to a 1.5 GB in-VM memory limit and 0.5 GB hypervisor cache limit for the container. Application performance for four different workloads, the Filebench webserver workload and YCSB [8] using Redis, MongoDB and MySQL data stores, with different allocation ratios is shown in Figure 5. Application throughput for Webserver and MongoDB remain almost unchanged (1000 ops/sec and 300 ops/sec, respectively) while Redis and MySQL performance degraded when more memory was split with the hypervisor cache. Redis was exceptional in extreme split scenarios—very high throughput (10406 ops/sec) was observed with 2GB VM memory allocated to the container and, application stall observed when 256 MB memory was allocated through the Cgroup and rest allocated in the hypervisor cache.

Explanation for the application behavior is presented in Table 1. Applications requiring anonymous memory can not be helped by the hypervisor cache and resort back to swapping as the last alternate. For applications depending on file I/O, hypervisor cache can offload the caching responsibility without causing any application degradation. These results show that providing the VM level memory manager with the option to choose different cache and in-VM memory configurations provides flexibility to meet various application SLA objectives.

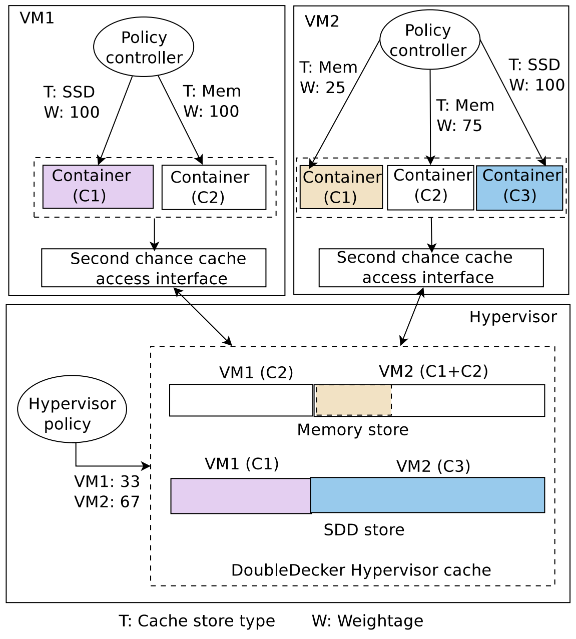


Figure 6. Proposed differentiated hypervisor cache partitioning across containers in a nested virtualization setting.

Existing centralized hypervisor cache management techniques (e.g., Morai [35], Centaur [21]) have limitations w.r.t. their applicability in such scenarios. The required flexibility to empower VM-level memory management across applications (as deduced from the previous experiment) is not supported. For example, existing techniques can neither improve performance of the applications that depend on anonymous memory allocations (e.g., Redis) nor provide flexibility to free in-VM memory by selectively offloading applications (e.g., Webserver) to the hypervisor cache. Additionally, existing hypervisor caching solutions [21, 35] operate in an inclusive cache operation mode w.r.t. the guest OS page cache which is wasteful due to duplicate storage of disk blocks in memory. Further, application differentiation at the hypervisor cache is not generic. For example, Morai [35] requires applications to be hosted on different virtual disks so that they can be distinguished at the hypervisor level.

To support differentiated policy enforcement in derivative cloud setups at the nested levels, a rethink of the hypervisor cache design is required. The proposed design should allow independent memory management at the two nested levels with the hypervisor still being the enforcing entity.

3 DoubleDecker Architecture

DoubleDecker provides a multi-layered differentiated cache partitioning and management solution. The software architecture of DoubleDecker is as shown in Figure 6. Its two main components are the second chance cache interface housed in each virtual machine and the DoubleDecker cache manager augmenting the hypervisor. The hypervisor cache is managed via two policy controllers, one to control per-VM provisioning and the other to control per-container provisioning in the corresponding virtual machines cache region. Our current design specifies allocations in terms of cache usage weightages in percentage between peer entities at each hosting level. Additionally, the policy controller in each virtual machine also specifies the cache store type of each container. A two tuple `<T, W>` is used for this purpose, where, `T` denotes the store type (in-memory or SSD) and `W` the weight specification for relative sizing.

Figure 6 shows a setup with two virtual machines with a cache allocation weightage percentage of 33 and 67. The per-VM ratio is applied to both the memory and the SSD store across all VMs [1]. The first virtual machine, VM1 hosts two containers and the second VM (VM2) hosts three containers. Virtual machine administrators use the local policy controller to specify distribution of the per-VM cache space across containers. For the example shown, the specification for VM1 for its two containers is <SSD, 100> and <Mem, 100>— Container 1 to be allotted all of VM1's share in the SSD store of the hypervisor cache and Container 2 to be allotted all of VM1's share in the memory store of the cache. Similarly, the specification for VM2 is to allocate its memory store in the ratio of 1:3 for the first two containers and allocate all of its share in the SSD store to the third container. Based on the above specifications the hypervisor memory store is shared by three containers and the SSD store by two. Simultaneously, the weightages for VM1 and VM2 for both the stores are 33 and 67, respectively.

3.1 Policy control mechanism

The Cgroup resource control framework for instantiating containers provides configuration options to specify allocation limits for different resources, e.g., CPU share, memory usage limit etc. per-cgroup. Additionally, for each container, the DoubleDecker policy controller requires specification of the hypervisor cache storage type and weight for cache sizing. An extension to the Cgroup resource control framework is warranted to incorporate this feature. Similar to the dynamic adjustment of resource limits, the hypervisor cache specifications are required to be handled in dynamic manner. Interfacing of the updated Cgroup resource control framework also requires extensions to the second chance cache access interface in the guest OS.

3.2 Second chance cache access interface

The second chance cache interface (explained in §2) assigns unique pool identifiers to each file system registered to use the second chance cache by delegating the call to the cache store implementation. The unique pool identifier is an integral part in the key used for operations like lookup, store etc.. In a derivative cloud setup several containers can be instantiated within a single file system, making file system based identifiers not applicable. Towards addressing this, extensions to the Cgroups subsystem and the second chance cache interface need to be integrated so that an unique identity is assigned to each application container for use by the hypervisor cache. When an application container boots up, the hypervisor cache interface is notified by the Cgroups subsystem, which in turn requests a new pool-ID from the hypervisor cache. Subsequent second chance cache accesses issued by the guest OS are accompanied by the container specific unique pool identifier. The unique pool-ID is derived by determining the container for the operation.

[1]A generalized setup where different cache size ratios are specified per-VM for the two stores is a straightforward extension.

3.3 Hypervisor cache store

The cache store implementation of DoubleDecker is required to support container-level policies apart from providing VM-level cache partitioning. The VM-level cache partition sizes are derived from ratios specified via the hypervisor-level policy controller. To apply application container level policies, configurations for containers are transmitted to the hypervisor cache manager via the guest OS cache interface. The hypervisor cache store applies the policies to partition the cache across the containers of a virtual machine and also maintains cache proportionality across virtual machine. The DoubleDecker solution provides backend storage implementation with support for dynamic reconfiguration of policies, both at the hypervisor-level and from within virtual machines. This feature is required for dynamic provisioning with changing workload characteristics of applications and to handle instantiation and removal of virtual machines and containers. With the derivative setup, a virtual machine can choose to store objects of all its containers in memory, or on the SSD or distributed over both in-memory and SSD storage on a per-container basis.

The hypervisor cache also implements an eviction procedure to flush objects in the cache during memory pressure. As part of this work, we implement a FIFO policy for evictions of objects on memory pressure. With two types of storage options available, a *trickle-down* and *trickle-up* cache is possible— objects evicted from the second-chance cache are stored in the third-chance cache. The design of DoubleDecker is centered around the idea that the semantics and decisions of how to manage the hypervisor cache should be left to the policy controller in the guest operating systems. This offers flexibility to the policy engine to explicitly size and choose the storage type for individual applications. For example, a policy engine can decide that an application that does mostly sequential reads or one whose working set size is very large, should have its objects stored in the SSD. Nevertheless, keeping the core principle of supporting cache management from within the VM, we provide a hybrid mode configuration option for the VM-level controller. In this mode, the VM-level controller can provision containers with both memory and SSD shares, where the SSD store is used only when the memory share of the container is exhausted. The comprehensive design and evaluation of the hybrid store is left as a future direction. In this work, we present results with applications configured to use either in-memory or SSD backed caches, not both.

4 Implementation

We implemented the DoubleDecker solution on the Linux+KVM virtualization platform and LXC [24] containers. Guest OS modifications are implemented in Linux virtual machines hosting LXC [24] containers. Linux `cleancache`, the second chance cache access interface is modified to implement the Cgroup extensions required for DoubleDecker

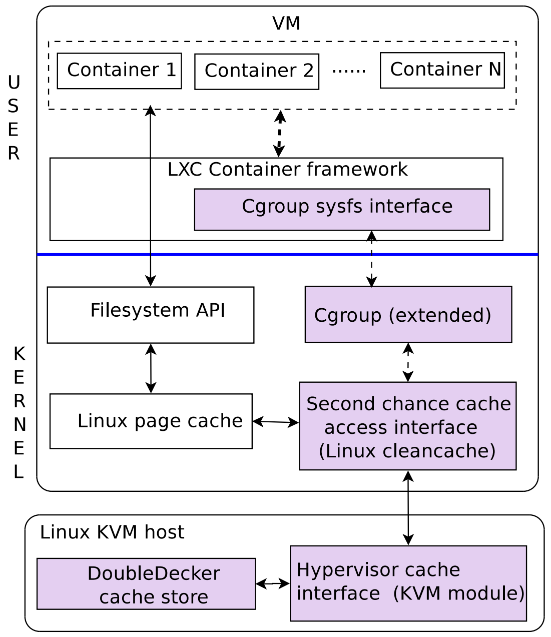


Figure 7. DoubleDecker implementation in KVM virtualized system. The shaded boxes in the figure represents modified/enhanced software components.

(Figure 7). Linux cleancache operations are routed to the KVM hypervisor through a new VMCALL [19]. As shown in Figure 7, the KVM hypervisor module is modified to capture this hypercall, copy the arguments on to the host memory and pass the call to the DoubleDecker hypervisor cache store.

4.1 Cgroup and cleancache modifications

Several modifications to the Linux Cgroups subsystem are required for container-level nested hypervisor cache management. Two new parameters have been added to the kernel state description of the Cgroups subsystem— (i) name of the Cgroups for which the hypervisor cache is to be enabled, (ii) policy configuration tuple (<`T, W`>) for each container to specify the storage type and weightage percentage. These parameters can be accessed as `sysfs` entries from the user space. Interactions between the Cgroup subsystem and the cleancache interface occur only if the Cgroup identifier matches the provided filter. The newly introduced second chance operations and the events triggering these are described below.

CREATE_CGROUP: When a new container is created, a new Cgroup kernel state is created and the cleancache interface is notified of the event. Linux cleancache is extended to handle a container creation event and which in turn forwards this event to the DoubleDecker hypervisor cache store. The DoubleDecker cache returns a new unique pool identifier (`pool-id`) corresponding to the newly created container. The pool-id is stored in the Cgroup kernel state of the guest OS and used for subsequent hypervisor cache operations.

SET_CG_WEIGHT: This event is generated when the Cgroup sysfs entry is updated (due to policy control decision) to update the container hypervisor cache specifications—the storage type and weightage percentage. Update to the Cgroup variables is captured by Linux kernel Cgroup module and passed on to the cleancache layer. The cleancache second chance cache interface transmits the specifications to the DoubleDecker hypervisor cache manager via KVM hypercall. The DoubleDecker cache manager updates its state for the container and updates state of the cache as necessary.

MIGRATE_OBJECT: Since the *key* used by hypervisor cache to index objects is Cgroup based and not based on file system information, a subtle issue arises due to mapping of files to application containers. Specifically, this is the case when Cgroup ownership for file blocks present in the hypervisor cache store changes from one Cgroup to another. This is usually the case when files are shared across application containers. To handle this condition, file blocks are migrated (mappings changed) from one hypervisor cache pool (corresponding to a container) to another.

DESTROY_CGROUP: When a container with a valid `pool-id` is shutdown, the cleancache interface is notified of the event. Linux cleancache is extended to handle a container destroy event and transmits the event to the DoubleDecker cache manager. The hypervisor cache frees up all the objects corresponding to the pool and marks the pool as free.

GET_STATS: It can be useful for the policy controller inside a VM to get per-container level cache allocation and cache usage related statistics for the containers executing in the VM. We have extended the cleancache interface to request the DoubleDecker cache for statistics when required by the policy controller.

With DoubleDecker, semantics of existing cleancache operations—lookup (get), store (put), invalidate (flush) (refer §2) remain the same, their implementation has the following modifications. The page cache layer passes a memory page along with the file inode number and block offset to the cleancache layer to enable second chance operations. With vanilla cleancache implementation, there is one-to-one correspondence between the `pool-id` and the file system superblock which can be extracted easily during cleancache operations. With DoubleDecker, the Cgroup owner is first deduced from the memory page to determine the unique `pool-id`. This is determined by finding the owner process for the page, and extracting the Cgroup entity to which the process belongs.

4.2 DoubleDecker hypervisor cache store

High-level design of DoubleDecker hypervisor cache store implementation is shown in Figure 8. The DoubleDecker cache manager interface is the entry point for all calls from the guest VMs (routed through the KVM hypervisor cache interface) and configuration changes by the host administrator.

The host administrator may configure the memory size limits and SSD device limits for the DoubleDecker cache store which is handled by the policy module. Further, container level configurations explained before is delegated to the policy module for cache management. On any configuration change, the policy module recalculates cache store entitlements at two levels—per-VM level and container (pool) level. The policy module monitors DoubleDecker cache usage by the VMs and containers and takes appropriate action (e.g., eviction) when the cache entitlement is violated. In the current

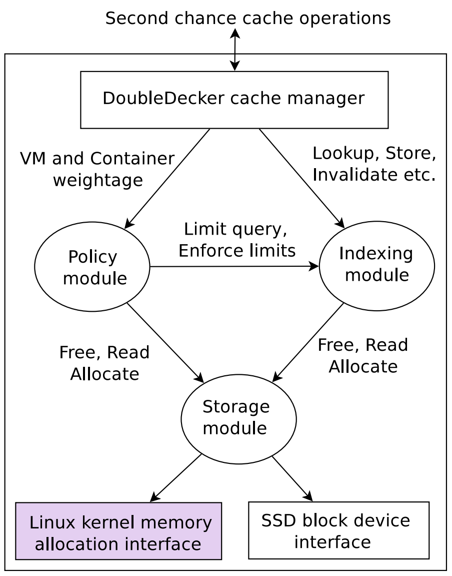

Second chance cache operations
DoubleDecker cache manager
VM and Container weightage
Lookup, Store, Invalidate etc.
Policy module
Limit query, Enforce limits
Indexing module
Free, Read Allocate
Free, Read Allocate
Storage module
Linux kernel memory allocation interface
SSD block device interface

Figure 8. DoubleDecker hypervisor cache components.

implementation, FIFO is used (LRU equivalent for exclusive caches) to implement eviction at a pool level.

The indexing module is responsible for implementing normal second chance cache operations like lookup (get), store (put) etc.. The key consisting of three tuples provided by the VM (<`pool-id`>, <`inode-num`>, <`block-offset`>) along with VM ID is used to map the request to storage objects. A hierarchy of indexing data structures—per-pool file object (`inode-num`) hash table, file block radix-tree etc.—are used to map the key to a storage object. The opaque storage object is exchanged with the storage module for actual storage operations. On a put request, the indexing module enforces the limits by checking the pool entitlement and evicting through the policy module, if necessary.

Storage module provides backend independent services to read storage blocks, allocate new storage blocks and free storage blocks. In the current DoubleDecker implementation, two types of storage backends are implemented. For memory storage backend, Linux kernel routines like `page_alloc`, `page_free`, `memcpy` etc. are used. For SSD backend implementation, we have implemented a raw block device IO layer on top of the generic block device driver interface. The read calls (for get operations) implements synchronous IO operations while write calls (for put operations) are implemented in an asynchronous manner.

4.3 DoubleDecker policy enforcement

To implement resource conservative cache management, cached blocks are evicted only when the DoubleDecker cache limit is reached. In such a case, selecting a victim application (container) is a two step process—first victim VM is selected and then the victim container in the selected VM is decided. Victim selection algorithm is same for both the levels (VMs and containers) and outline of the algorithm is presented in Algorithm 1. The algorithm takes list of entities (VMs or Containers) and eviction size as input to return the victim entity. List of entities who are over the limit of their

Algorithm 1 Victim selection from a list of cache using entities (VMs or Containers)

```
1: procedure GETVICTIM($Entities[1..n]$, $EvictionSize$)
     ▷ $EvictionSize$: # of cached blocks to be evicted
     ▷ $Entities$: List of entities (VMs or containers)
2:     $overusedlist[1..n]$
3:     $cumlweight = 0$
4:     $underusedbuf = 0$
5:     $count = 0$
6:     for i=1 to n do
7:         $E_i = Entities[i]$
8:         if $E_i.entitlement < E_i.used + EvictionSize$ then
9:             $overusedlist[count] = E_i$
10:            $cumlweight$ += $E_i.weightage$
11:            $count$++
12:        end if
13:        if $E_i.entitlement - E_i.used > 2 * EvictionSize$ then
14:            $underusedbuf$ += $E_i.entitlement - E_i.used$
15:        end if
16:    end for
17:    $E = overusedlist[1]$
18:    $exceedmax$ = exceed($E$, $underusedbuf$, $cumlweight$)
19:    for i=2 to count do
20:        $E_i = overusedlist[i]$
21:        if $exceedmax <$ exceed($E$) then
22:            $E = E_i$
23:            $exceedmax$ = exceed($E$)
24:        end if
25:    end for
26:    Return E
27: end procedure
```

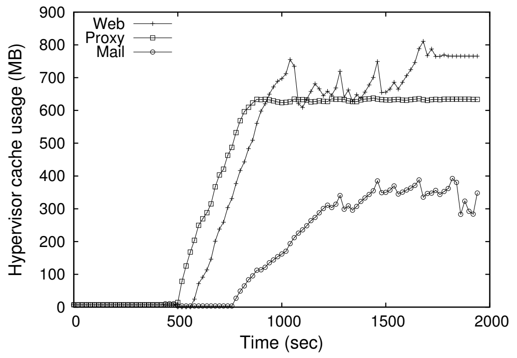
entitlement—calculated directly applying the DoubleDecker cache weightage configuration—are determined (line# 8-12) and stored in the *overusedlist*. Sum of underutilized memory (*underusedbuf*) is redistributed among the entities as per their weightages to calculate their effective entitlements (*E.entitlement* + (*b* * *E.weightage* / *cw*)) and exceed values . The exceed value for an entity is calculated as,

$$exceed(E, b, cw) = E.used + EvictionSize - (E.entitlement + (b * E.weightage/cw)) \quad (1)$$

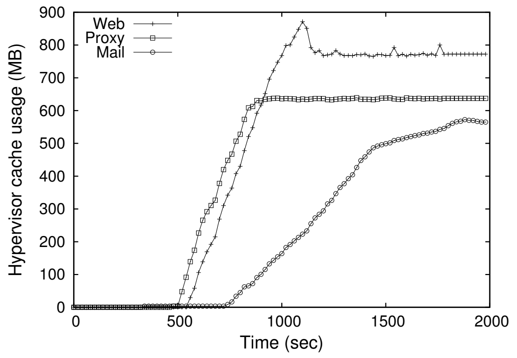
where b is the sum of under utilized entitlements (*underusedbuf* in Algorithm 1), *cw* is the sum of weightage percentage of entities whose cache usage is above their respective entitlements. The entity calculated with the highest exceed value is selected as the victim for eviction. Only one victim entity is selected because we use a small batch (2MB) for eviction when a store request can not be serviced because of global limit violations.

5 Experimental evaluation

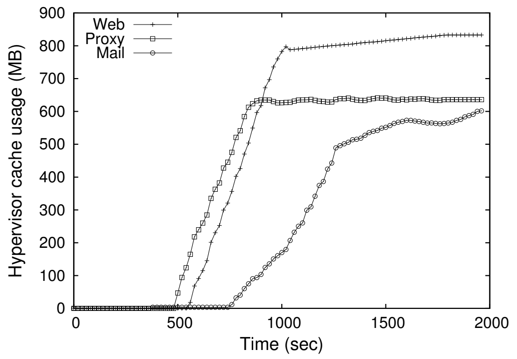
The efficacy of DoubleDecker is established with a set of experiments using the following setup, a 16-core blade server with 3.4 Ghz CPUs (Intel Xeon CPU E5-2650) and 32 GB RAM. A 240GB Kingston Digital SSDNow V300 SATA 3 solid state disk, accessed through the SATA interface. As



(a) Global hypervisor cache (memory backed)



(b) DoubleDecker hypervisor cache (memory backed)



(c) DoubleDecker SSD backed hypervisor cache

Figure 9. Hypervisor cache distribution across application containers with different cache settings.

workloads, we used the Webserver, Proxycache, Mail and Videoserver profiles of the Filebench [36] workload suite.

5.1 Impact of caching modes

The aim of this experiment is to establish correctness of the two different cache store options of DoubleDecker, and to demonstrate the benefits of partitioning the hypervisor cache.

5.1.1 Cache size distribution

For the experiment, a VM with 8 VCPUS and 8 GB RAM was used. Four application containers (Container 1-4) configured with 1GB RAM each, executed the Webserver, Proxycache, Mail, Videoserver workloads. All other resources (CPU and

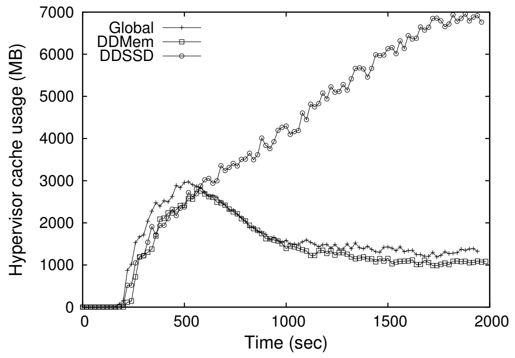


Figure 10. DoubleDecker cache usage by Videoserver-workload with different cache configurations.

network) were allocated equally across the application containers. Three hypervisor caching modes were compared—(i) a memory backed hypervisor cache with capacity 3 GB with global cache management mode, no partitioning on a per-container basis (referred as *Global*) (ii) a 3 GB memory backed hypervisor cache with DoubleDecker cache management partitioning the cache equally among all containers (referred as *DDMem*), (iii) a 240 GB SSD backed hypervisor cache with DoubleDecker cache management partitioning the cache equally among all containers (referred as *DDSSD*),

Hypervisor cache distribution across the containers executing the Webserver, Proxycache and Mail workloads with different caching configurations is shown in Figure 9. The Videoserver workload dominated the cache usage; for better presentation cache usage of the Videoserver workload with different caching modes is shown in Figure 10. As can be seen from Figure 9 and Figure 10, for the first 500 seconds there is no requirement of the hypervisor cache except for the Videoserver workload. During this duration, container with the Videoserver workload occupies up to 3 GB of the cache (in all caching modes). Beyond 500 seconds, the memory requirements of other workloads also do not fit in the VM and the corresponding containers contend for the second chance cache. Correspondingly, the Videoserver workloads cache occupancy decreases from 3 GB to 1.4 GB.

With the global cache policy, there is no deterministic cache capacity for each container and all four workloads contend for the cache. The result cache size distribution for each workload is dependent on its access patterns and the access rate. Since Videoserver workload has the higher IO rate and quantity, it still consumes the largest portion of the cache. The cache of the Webserver and Mail workloads gets affected the most, as objects of these containers are evicted from the cache due to pressure from the other two workloads. In fact, the Mail workload gets a maximum share of less than 400 MB, as against its fair share of 750 MB.

With DoubleDecker equal-weight cache partitioning (Figure 9b), the Videoserver-workload was allocated 1.2 GB when other workloads started using the cache up to their own entitlements. Since, the Proxycache and Mail workloads did not consume their entire share of 750 MB, the remaining capacity was shared between the Videoserver and Webserver workloads. In fact the Webserver workload does not consume

	Global (Memory)				DoubleDecker (Memory)				DoubleDecker (SSD)			
Workload	Throughput (MB/s)	Latency (ms)	lookup-to store ratio (%)	# of evic--tions	Throughput (MB/s)	Latency (ms)	lookup-to store ratio (%)	# of evic--tions	Throughput (MB/s)	Latency (ms)	lookup-to store ratio (%)	# of evic--tions
Webserver	14.2	29.2	93	209815	93.7	3	99	0	5.5	73	91	0
Proxycache	5.4	89.1	76	2880	5.6	85.9	76	0	5.2	91.5	75.9	0
Mailserver	1.3	598.2	1	159793	1.4	555.7	32	0	2	386.7	44	0
Videoserver	1276	2.6	65	1650424	1188	2.9	57	2076672	481.5	7.8	67	0

Table 2. Application performance and cache behavior comparison with different hypervisor caching schemes.

Cache Setting	Webserver (C1)	Proxycache (C2)	Mail (C3)	Videoserver (C4)
DDMem	Mem: 32	Mem: 25	Mem: 25	Mem: 18
DDMemEx	Mem: 40	Mem: 30	Mem: 30	Mem: 0
DDHybrid	Mem: 40	Mem: 30	Mem: 30	SSD:100

Table 3. DoubleDecker cache configuration settings.

more than 800 MB and hence 1.2 GB is consumed by the Videoserver workload. Note that with DoubleDecker mode, cache usage of workloads other than Videoserver-workload did not take any dips once they reached their respective peaks as opposed to global mode. This demonstrates *resource conservative nature* of DoubleDecker cache provisioning with *guaranteed isolation* according to assigned priorities. With SSD backed hypervisor caching (Figure 9c), the cache size was sufficient to for all the workloads. This verified two things, one the DoubleDecker could manage a SSD-based cache store correctly and second the peak cache usage in the DDMem caching mode.

5.1.2 Performance impacts with caching modes

The impact of cache distribution with different caching modes is reported in Table 2. Webserver throughput with the DDMem caching mode (DoubleDecker with in-memory cache store) was approximately *six times* better than the global caching mode. With the DoubleDecker DD-mem caching mode, Mail and Proxycache workloads resulted in marginal improvements (∼5%) and the Videoserver-workload resulted in small degradation (∼6%) in application performance. With the global eviction mode, evictions from workloads other than Videoserver were noticed, e.g., ∼200K evictions from the Webserver container. With DoubleDecker, only the Videoserver-workload was victimized to enforce equity at the container-level according to the equal cache distribution configuration.

No evictions were noticed with the SSD-backed hypervisor cache as the cache could host all disk cache blocks for all the containers. Because of the increased IO latency of SSD access, application throughput for the Webserver and Videoserver workloads was around 300% lower compared to memory backed hypervisor cache. Interestingly, throughput and latency of Mail workload improved by 30% and 40%, respectively. This is due to offloading of disk operations of other workloads through sufficient SSD availability in the hypervisor cache. These results show that even in case of sufficient hypervisor cache availability, enforcing fairness across applications can result in significant application benefits.

5.2 Flexible hypervisor cache management

To analyze the effectiveness of container level priority extensions to the hypervisor caching, we have performed an experiment with application containers configured with different memory limits. In this experiment, the Webserver container (C1) was configured with a memory limit of 1.25 GB (through Cgroups) and the Videoserver container (C4) was configured with a memory limit of 750 MB. Other two containers were configured with 1 GB memory limit each. The DoubleDecker memory cache size was limited to 2 GB. In the global eviction mode (referred to as Global), only the container level memory limits were in action while the DoubleDecker hypervisor cache was shared globally without any container-level priority enforcement. Three different cache settings (DDMem, DDMemEx and DDHybrid), shown in Table 3, were used to partition the DoubleDecker hypervisor cache across the application containers. The DDMem policy extended the Cgroup level memory allocation weights to the DoubleDecker hypervisor cache. The DDHybrid policy used the SSD store for the Videoserver workload, while containers C1-C3 shared the memory cache with weights 40, 30 and 30, respectively.

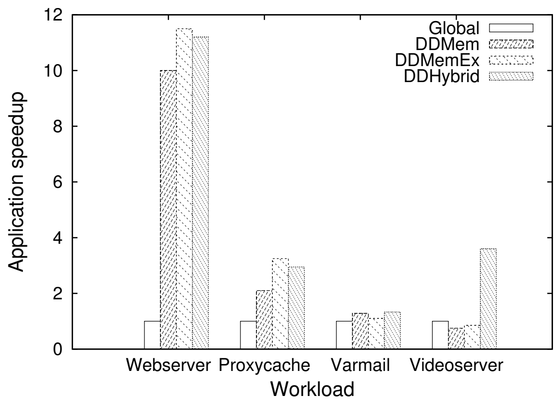
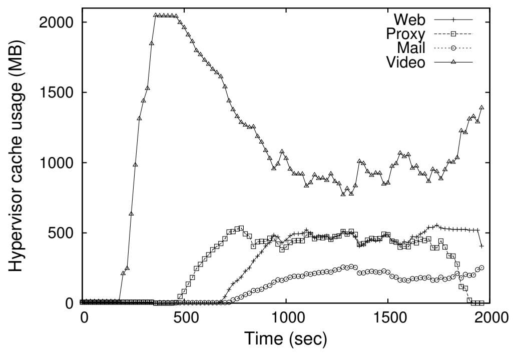
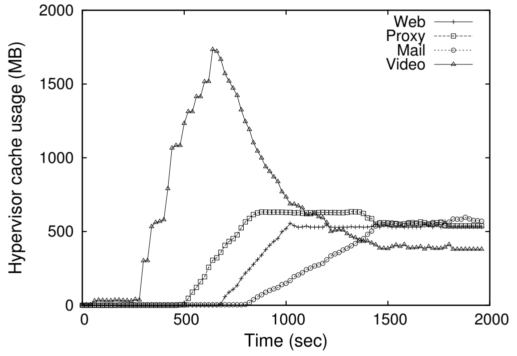


Figure 11. Comparison of application performance with differentiated hypervisor caching policies vs. global hypervisor cache management.

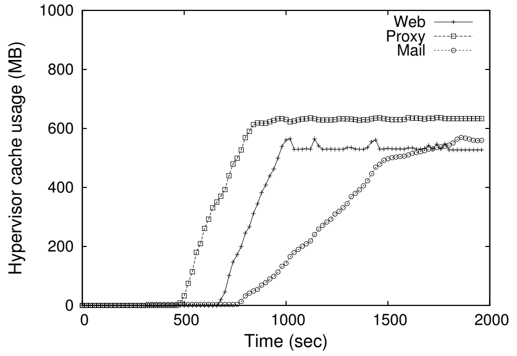
Application throughput improvements w.r.t the global hypervisor cache management mode and the different DoubleDecker caching policies is shown in Figure 11. For the Webserver-workload, significant application throughput improvement was observed—10x, 11x and 11x compared to global caching with DDMem, DDMemEx and DDHybrid policies, respectively. Around 2x, 3.2x and 3x application performance improvement resulted for the Proxycache-workload with the three DoubleDecker caching modes. With the Mail workload, the application performance gains were marginal. With the Videoserver-workload application performance decreased with the DDMem (∼25%) and DDMemEx (∼20%) caching modes compared to that of the Global mode. This was because the application's aggressive hypervisor cache usage behavior was curtailed by DoubleDecker policies. When the Videoserver-workload's hypervisor cache was moved to



(a) Global hypervisor cache



(b) DDMem DoubleDecker policy



(c) DDHybrid DoubleDecker policy

Figure 12. Hypervisor cache distribution across application containers with different cache management policies.

the SSD, 3.6x improvement in application throughput was observed compared to the global mode caching mode.

Hypervisor cache distribution across application containers with the different caching modes as shown in Figure 12. With the global mode, the hypervisor cache usage was dominated by the Videoserver-workload while other applications (C1-C3) contended for the cache and could never reach a proportionate sharing ratio. WithDoubleDecker's DDMem policy, cache usage of Videoserver-workload reduced to the minimum (around 400 MB), close to its fair share, when other applications started using the cache. Memory cache usage by the Webserver, Proxycache and Mail workloads is shown in Figure 12c when hypervisor cache for the Videoserver-workload was moved to the SSD cache (the DDHybrid mode). In this case, the 2 GB memory available for the memory backed region of the DoubleDecker hypervisor cache was sufficient to serve all the application without resulting in any evictions. The three containers, each occupied about 500 MB to 600 MB of the cache.

Workload *(SLA)*	Technique	Throughput (ops/sec)	App. memory, Hypervisor cache (GB)
MongoDB	Morai++	16.9	0.5, 1.3
(15 ops/sec)	DoubleDecker	25.1	1, 0.4
MySQL	Morai++	48.5	1.6, 0
(100 ops/sec)	DoubleDecker	132.7	1.9, 0
Redis	Morai++	13	1.8, 0
(5000 ops/sec)	DoubleDecker	11186	2, 0
Webserver	Morai++	1289	2.2, 0.5
(900 ops/sec)	DoubleDecker	988	1, 1.6

Table 4. Comparison of centralized second chance cache management with DoubleDecker.

The main take-away from this experiment was that usage of the hypervisor cache for different applications can be configured carefully, either to use the memory store or the SSD store, for positive benefits for all applications.

5.2.1 Efficacy of cooperative memory management

DoubleDecker, with its cooperative memory management framework, can explore holistic provisioning solutions that traditional centralized hypervisor based techniques cannot. To demonstrate such capabilities, we performed the following experiment. A single VM hosted four application containers and executed the MongoDB, Redis, MySQL data stores and the Filebench webserver workload. The data stores acted as backends for YCSB clients. The VM was provisioned with 6GB memory and 8 CPUs (2 CPUs pinned to each application container). The hypervisor cache was provisioned to 2GB. To approximate a centralized hypervisor cache management approach like Morai [35], we iterated over different cache partitions at the hypervisor cache and report results of the best configuration (referred to as Morai++). Each application had a target throughput SLA to achieve (mentioned in the first column of Table 4). The best configuration was the one which met the individual application SLAs and yielded the maximum aggregate throughput.

Referring to Table 4, a hypervisor cache partition of 60:40 between the MongoDB workload and the Webserver workloada met their SLA requirements and yielded the maximum aggregate throughput with Morai++. Morai++ could not satisfy application performance targets of Redis and MySQL workloads while barely managed to meet MongoDB performance targets. In this setup, usage of the 6GB VM memory (shared by all containers) was 0.5 GB, 1.6 GB, 1.8 GB and 2.2 GB, for the MongoDB, MySQL, Redis and Webserver workloads, respectively. Interestingly, with Morai++, actual hypervisor cache usage was not in the ratio of 60:40 and total average usage was below 2 GB (1.8 GB). This was because the Webserver workload dominated the in-VM memory usage and did not need its maximum share of hypervisor cache leaving MongoDB-workload to use the residual amount as per its demand (See Algorithm 1). The MongoDB workload (throttled inside the VM) could not leverage the maximum

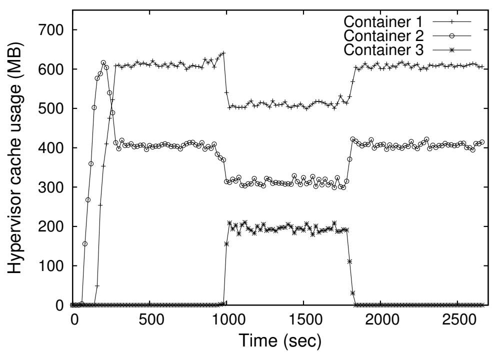


Figure 13. Dynamic policy changes with DoubleDecker cache and its impact on cache distribution across containers.

possible hypervisor cache usage. As an additional effect, the in-VM memory usage dominance of the Webserver-workload adversely affected other applications, notably Redis whose working set size was 2 GB and which did not fit in memory. Note that, both Redis and MySQL applications are designed to work only with anonymous memory and did not use the hypervisor cache integrated with the file IO path. With DoubleDecker, the VM-level manager can incorporate this information to provision memory both from the VM and the hypervisor cache which the centralized hypervisor caching techniques lack. In our experiment, with DoubleDecker, we configured the VM memory to be allocated as 1 GB, 2 GB, 2 GB and 1 GB, for the MongoDB, MySQL, Redis, and Webserver workloads, respectively and discovered the best hypervisor cache partition. Based on this setup, DoubleDecker meets the application SLA requirements of all the applications and the aggregate throughput achieved is higher by a large margin. This achieved primarily through improvements in the throughputs achieved by the MySQL and Redis applications. Note that, both these memory-bound applications were positively benefited by DoubleDecker. Also, since the working set of Redis could be accommodated in memory, its throughput increased by a factor of 1000.

Based on the above results, we demonstrate that two-level provisioning capability of DoubleDecker enables holistic memory configurations which can account for application characteristics and requirements, and yield improved application-level and system-wide performance. For workload provisioning in an adaptive manner, DoubleDecker can employ well known techniques like MRC [45, 46], WSS estimation [10, 44], SHARDS [40] etc.. Note that, the estimation should be done from within the VM which allows the guest OS memory manager to provision memory resources at the two levels. Centralized nesting agnostic hypervisor cache management techniques cannot explore such provisioning configurations.

5.3 Dynamic cache management

To demonstrate DoubleDecker's capability to apply dynamic hypervisor cache partitioning across containers and virtual machines, we performed the following experiments.

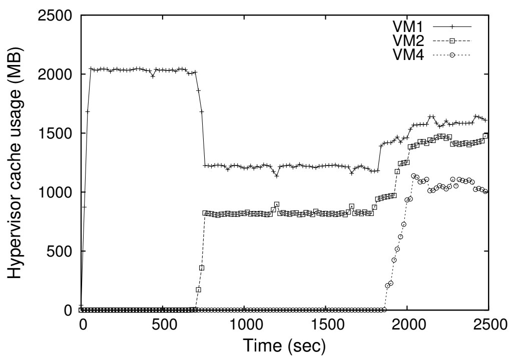


Figure 14. Dynamic VM provisioning with policy changes triggering DoubleDecker cache redistribution across VMs.

5.3.1 Dynamic management across containers

Initially, a single VM was configured with two application containers. A 1 GB memory limit was configured for DoubleDecker memory cache while the SSD cache size was 240 GB. Container 1 and 2 were configured with 1 GB memory each in the virtual machine and executed the Webserver and Proxycache workloads, respectively. The DoubleDecker cache distribution weights for Container 1 and 2 were set to 60 and 40, respectively. After 900 seconds into the experiment, a third container (Container 3) was booted which executed the Videoserver-workload. At this point, the DoubleDecker cache distribution weights were configured as 50, 30 and 20 for Containers 1-3, respectively. After continuing with this setting for 900 seconds, the Videoserver-workload (Container 3) was configured to use the SSD store for its hypervisor cache and the share of Container 1 and 2 for the memory store was reset to 60 and 40, respectively.

The DoubleDecker memory cache allocation across the three containers is as shown in Figure 13. Initially, the cache allocation for Containers 1 and 2 were approximately 600 MB and 400 MB, respectively. At around 1000 seconds from the start of the experiment, the DoubleDecker cache got redistributed—∼500 MB for Container 1, ∼300 MB for Container 2 and ∼ 200 MB for Container 3—when Container 3 started using its cache entitlement. Finally, when the DoubleDecker cache setting for Container 3 was dynamically changed from memory to SSD (at around 1800 seconds), the hypervisor memory cache was re-distributed in the ratio of 60:40 between Container 1 and Container 2. Container 3 used the SSD backed DoubleDecker cache for the rest of the experiment (not shown in Figure). DoubleDecker's support for dynamic cache partitioning policies provides flexibility in designing higher level memory management solutions.

5.3.2 Dynamic management across virtual machines

In a dynamic VM provisioning setup, the number of VMs and resources available for each change with time. To demonstrate DoubleDecker's suitability in this setup, we performed the following experiment. Four VMs, each configured with 4GB RAM were started at an interval of 600 seconds and each executed the Videoserver workload within a single application container. Initially, the DoubleDecker memory backed cache size limit was 2 GB and VM1 was given a weightage of 100. When VM2 boots up (at 600 seconds), the DoubleDecker cache weights for VM1 and VM2 were set to 60 and 40, respectively. VM3 was only allowed to use the SSD cache when it started at around 1200 seconds; VM1 and VM2 cache settings remained the same. At around 1800 seconds from the start of the experiment, another virtual machine VM4 was started. and the DoubleDecker memory backed cache size was increased to 4 GB. Further, the cache distribution weightages for VM1, VM2 and VM4 were configured as 40, 35 and 25, respectively.

DoubleDecker cache distribution across VM1, VM2 and VM4 is as shown in Figure 14. VM1 used up the DoubleDecker cache entirely (2 GB) until VM2 started using the cache (at around 650 seconds) when the cache share of VM1 and VM2 became ∼1200 MB and ∼800 MB, respectively. Instantiation of VM3 (at around 1800 seconds) did not disturb the cache distribution between VM1 and VM2 as VM3 used the SSD cache (VM3 usage not shown in Figure). Finally, when VM4 started and the cache capacity of DoubleDecker was changed along with cache partitioning weights, average cache usage of VM1, VM2 and VM4 was around 1600 MB, 1400 MB and 1000 MB, respectively.

With these two experiments, we demonstrated the capability of DoubleDecker to dynamically manage two-levels of cache specifications in a dynamic manner. Policy decision at the hypervisor which effect VM-level cache partitioning and decisions that effect the per-container partition sizes for each VM can be handled by DoubleDecker in a dynamic manner. This feature is a key enabler to develop adaptive memory and cache management policies depending on application behavior, workloads and execution entity priorities.

6 Related work

Hypervisor caching: A hosted virtualization solution like KVM [20, 25] naturally offers *inclusive* hypervisor caching by virtue of being in the disk access path. Singleton [33] performs cache translation of the inclusive hypervisor cache to provide exclusive caching by deduplicating KSM scanned pages and the inclusive cache. Transcendent memory [28] proposed a guest OS supported exclusive caching framework which and was first implemented in Xen [27]. Exclusive hypervisor cache for KVM [30] based on the transcendent memory model enabled several caching combinations and analyzed effectiveness of the hypervisor caching solutions. All of the above solutions provide hypervisor cache management capability at a VM granularity whereas DoubleDecker extends the framework to support application level differentiated hypervisor cache partitioning in derivative cloud setups. Mortar [18] exposed the hypervisor cache to distributed memory store applications like memcached [12] by modifying the application in an explicit manner. Our approach is more generic and does not require any modifications to applications. Software-defined caching [35] proposed application SLA guided dynamic second chance inclusive cache provisioning by hosting applications on separate virtual disks. DoubleDecker provides additional flexibility in terms of hypervisor cache provisioning with exclusive cache support and by providing different second chance storage options. Moreover, our approach is better equipped to meet the decentralized memory management requirements of a derivative cloud setup as shown in §2.3.1 and §5.2.1.

Application containers and derivative clouds: Linux Cgroups [22], BSD jails [16] and Solaris Zones [4] provide operating system level isolation support for different applications. LXC [24] and Docker [11] are two popular application container enablers built by exploiting the Linux Cgroup resource control framework. Nested virtualization [5, 14, 41] enables hosting VMs inside a VM by exposing hardware virtualization features to the first level VM. The goal of nested virtualization has been to address security issues [13], VM standardization related problems and hypervisor troubleshooting. Resource management in nested virtualization setups has been proposed by [6] and [42]. Intercloud [6] and Supercloud [42] propose interoperable cloud services decoupled from the cloud provider. Derivative clouds in the current era [17, 31, 34] use containers inside the VMs as the IaaS primitive because virtualization overheads using containers is significantly lower than nested VM virtualization. DoubleDecker enhances the provisioning flexibility through hypervisor cache management at the nested container level and opens up resource-based SLA business model enhancements for derivative clouds.

Memory management in virtualized systems: Memory ballooning [32, 39] facilitates dynamic adjustment of VM memory allocations. Several dynamic ballooning based controllers to estimate and adjust the memory allocation to the VMs have been proposed [7, 10, 45]. Hypervisor caching and DoubleDecker complements the dynamic memory controllers by providing symbiotic resource management capabilities. DoubleDecker furthers the flexibility in memory resource management in container within VM setups. Memory deduplication [1, 29, 39] is another memory efficiency enhancement technique shown to be useful in virtualized systems [1, 3]. Integration of DoubleDecker with deduplication to enhance the memory management efficiency in a derivative cloud setup would be an interesting future direction of our work.

7 Conclusions

Popularity of low-overhead application container frameworks has enabled derivative cloud based business models. Several challenges in efficient resource management for nested (containers inside VMs) setups require rethink of existing designs. In this paper, we proposed DoubleDecker, a cooperative memory management framework which enabled multi-level provisioning of VM-level memory and the hypervisor cache. DoubleDecker supports two storage backends i.e., memory and SSD, and provisioning configurations at two levels—per-VM and per-application container, for adaptive provisioning. Our experimental evaluation demonstrated the effectiveness of DoubleDecker by extending container level priorities to the hypervisor cache. We also showed that selective usage of different storage options of the hypervisor cache based on application characteristics, improved overall performance in the derivative setup. Most importantly, DoubleDecker can simultaneously provision memory at the VM-level and the hypervisor cache, and effectively meet requirements of decentralized memory management in derivative cloud setups.

References

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

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *SIGOPS Operating Systems Review 37*, 5 (Oct. 2003), 164–177.

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

[4] Beck, J., Comay, D., Ozgur, L., Price, D., Andy, T., Andrew, G., and Blaise, S. Virtualization and namespace isolation in the solaris operating system (psarc/2002/174), 2006.

[5] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour, B.-A. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), pp. 423–436.

[6] Bernstein, D., Ludvigson, E., Sankar, K., Diamond, S., and Morrow, M. Blueprint for the intercloud - protocols and formats for cloud computing interoperability. In *Proceedings of the Fourth International Conference on Internet and Web Applications and Services* (2009), pp. 328–336.

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

[8] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), pp. 143–154.

[9] Corbet, J. zcache: a compressed page cache. www.lwn.net/Articles/397574/. Accessed: 2016-11-18.

[10] Denning, P. J. The working set model for program behavior. *Communications of ACM 11*, 5 (1968), 323–333.

[11] Docker. Docker. https://www.docker.com/. Accessed: 2016-11-18.

[12] Fitzpatrick, B. Distributed caching with memcached. *Linux J. 2004*, 124 (2004).

[13] Ford, B., and Cox, R. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical conference* (2008), pp. 293–306.

[14] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels meet recursive virtual machines. *SIGOPS Operating Systems Review 30*, SI (1996), 137–151.

[15] Google. Google cloud platform. https://cloud.google.com/container-engine. Accessed: 2016-11-18.

[16] henning Kamp, P., and Watson, R. N. M. Jails: Confining the omnipotent root. In *Proceedings of the second International SANE Conference* (2000).

[17] Heroku. The heroku platform. https://www.heroku.com/platform. Accessed: 2016-11-18.

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

[19] Intel. Intel 64 and ia-32 architectures developer's manual: Vol. 3b. www.intel.com, 2016.

[20] Kivity, A. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium* (July 2007), pp. 225–230.

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

[22] Linux. Cgroup. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. Accessed: 2016-11-18.

[23] Linux. Kernel documentation: vm/cleancache. www.kernel.org/doc/Documentation/vm/cleancache.txt. Accessed: 2016-11-18.

[24] Linux. Linux containers. https://linuxcontainers.org/. Accessed: 2016-11-18.

[25] Linux. Tuning kvm. http://www.linux-kvm.org/page/Tuning_KVM. Accessed: 2016-11-18.

[26] Lu, P., and Shen, K. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of the USENIX Annual Technical Conference* (2007), pp. 3:1–3:15.

[27] Magenheimer, D. Update on transcendent memory on xen. Xen Summit 2010, 2016.

[28] Magenheimer, D., Mason, C., McCracken, D., and Hackel, K. Transcendent memory and linux. In *Proceedings of Linux Symposium* (2009), pp. 191–200.

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

[30] Mishra, D., and Kulkarni, P. 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.

[31] PI. Pi cloud services. https://www.picloudservices.com/. Accessed: 2016-11-18.

[32] Schopp, J., Fraser, K., and Silbermann, M. Resizing memory with balloons and hotplug. In *Proceedings of Linux Symposium* (2006), pp. 313–319.

[33] Sharma, P., and Kulkarni, P. 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.

[34] Sharma, P., Lee, S., Guo, T., Irwin, D., and Shenoy, P. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), pp. 16:1–16:15.

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

[36] Sun. Filebench. www.filebench.sourceforge.net/wiki/index.php/Main_Page. Accessed: 2016-11-18.

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

[38] VMware. The Role of Memory in VMware ESX Server 3. http://www.vmware.com/pdf/esx3_memory.pdf, 2016.

[39] Waldspurger, C. A. Memory resource management in vmware esx server. *SIGOPS Operating Systems Review 36*, SI (Dec. 2002), 181–194.

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

[41] Williams, D., Jamjoom, H., and Weatherspoon, H. The xen-blanket: Virtualize once, run everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), pp. 113–126.

[42] Williams, D., Jamjoom, H., and Weatherspoon, H. Plug into the supercloud. *IEEE Internet Computing 17*, 2 (2013), 28–34.

[43] Wood, T., Tarasuk-Levin, G., Shenoy, P., Desnoyers, P., Cecchet, E., and Corner, M. D. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 5th international conference on Virtual Execution Environments* (2009), pp. 31–40.

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

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

[46] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th conference on Architectural support for programming languages and operating systems (ASPLOS)* (2004), pp. 177–188.