

# Co-operative Hypervisor Caching based on Content Similarity

*A Thesis*

*submitted in partial fulfillment of the  
requirements for the degree of  
Master of Technology*

*by*

**Aby Sam Ross  
143050093**




Department of Computer Science & Engineering  
Indian Institute of Technology Bombay  
Mumbai 400076 (India)

August 2016

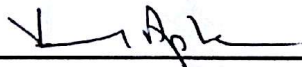
## Dissertation Approval

This dissertation entitled “Co-operative Hypervisor Caching based on Content Similarity”, submitted by Aby Sam Ross (Roll No: 143050093) is approved for the degree of Master of Technology in Computer Science and Engineering from Indian Institute of Technology Bombay.



---

Prof. Purushottam Kulkarni  
Dept. of CSE, IIT Bombay  
Supervisor



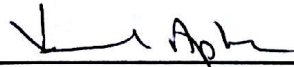
---

Prof. Varsha Apte  
Dept. of CSE, IIT Bombay  
Internal Examiner



---

Mr. Bhaskaran Raman  
Dept. of CSE, IIT Bombay  
Internal Examiner



---

Chairperson

## Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date: 11<sup>th</sup> AUGUST 2016

Place: IIT Bombay, Mumbai



Aby Sam Ross

Roll No: 14350093

## **Acknowledgement**

I would like to express my sincere gratitude to Prof. Purushottam Kulkarni for hand holding me into the world of virtualization, believing in me while letting me take up this project and during it's course. For if it were not for these the completion of this thesis would have been impossible. I would also like to thank my lab/batch mates from whom I got to learn a lot.

**Aby Sam Ross**  
**Roll No: 143050093**

## **Abstract**

The aim of memory management techniques in virtual environment is to optimize, across time, the distribution of machine memory (RAM) among a maximal set of virtual machines (VMs). Hence over-commitment of memory becomes inevitable to achieve this lofty goal. When memory is being over-committed efficient management of memory becomes a necessity to ensure smooth running of the show. There are many techniques for managing memory in an over-committed scenario. E.g demand paging or ballooning based dynamic memory provisioning, duplicate content elimination etc.

But previous work have shown that there is no ‘one-size-fits-all’ technique among these that addresses the different challenges over-commitment throws at us. In fact using just any one of these as the lone management policy can prove counter productive and negatively impact the performance of applications running in virtual machines. This is because these techniques can end up adversely affecting existing memory management techniques of native OS in the VM or applications that manage their own memory.

This thesis explores a relatively new virtualized memory management paradigm called hypervisor caches which, when provisioned as a second chance page cache, mitigates the negative effects of traditional management techniques like ballooning on applications. A second chance hypervisor cache does this by caching pages that were evicted from a VM experiencing memory pressure due to balloon inflation. In fact such a hypervisor cache if combined with ballooning and memory sharing has the potential to become a holistic and efficient memory management policy. Specifically this thesis tries to exploit the presence of similar content in data centres, the fact that many data centres have their secondary storage on network and the presumption that accessing memory over a network should be faster than accessing the disk across the same. This thesis implements a second chance, disk page caching, hypervisor cache that pursues content de-duplication opportunities within itself and in other hypervisors. It also profiles the run-times of individual operations involved in the implementation and finds out the overheads of having such a cache on the end-to-end delays experienced by a read intensive synthetic workload.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Issues with memory usage and management in virtualized environment .	2
1.2	Hypervisor Caching: Another memory management technique . . . . .	5
1.3	Motivation . . . . .	6
1.4	Problem Statement . . . . .	8
1.5	Outline of the Report . . . . .	8
<b>2</b>	<b>Background</b>	<b>9</b>
2.1	Transcendent memory basics . . . . .	9
2.2	Tmem use cases . . . . .	11
2.3	Tmem backends . . . . .	14
<b>3</b>	<b>Related Work</b>	<b>17</b>
3.1	Determining memory usage statistics & finding an optimal balloon size .	17
3.2	Content based de-duplication . . . . .	18
3.3	Virtualized memory management for applications managing own memory	22
3.4	Hypervisor managed cache . . . . .	23
<b>4</b>	<b>Design — local de-duplication</b>	<b>24</b>
4.1	Design steps . . . . .	24
4.2	Design inspirations . . . . .	24
4.3	Tmem backend with local de-duplication — design & implementation . .	25
<b>5</b>	<b>Design — co-operative de-duplication</b>	<b>32</b>
5.1	Challenges of doing co-operative de-duplication . . . . .	32
5.2	Co-operative or Remote de-duplication — design & implementation . . .	33

---

5.3	Summary . . . . .	46
<b>6</b>	<b>Experiments &amp; Evaluations</b>	<b>48</b>
6.1	Experimental Setup . . . . .	48
6.2	Correctness Verification . . . . .	48
6.3	Evaluation of runtime overheads . . . . .	53
6.4	Micro-benchmarking of important functions in the backend implementation	54
<b>7</b>	<b>Extensions &amp; Conclusions</b>	<b>58</b>

# List of Tables

1.1	Effectiveness of G-GT compared to G-H for read intensive workloads . . .	6
2.1	Different types of tmem pools. . . . .	10
2.2	Advantages of tmem to different use cases . . . . .	11
6.1	Effects of hot & cold tmem caches in local de-dup only mode on a synthetic workload in a VM hosted in machine B. Runtime is specified in seconds. Base case mean 14.456s. The results were obtained over 100 runs.	53
6.2	Effects of hot & cold tmem caches in local & remote de-dup mode on a synthetic workload in a VM hosted in machine B. Runtime is specified in seconds. Base case mean 14.456s. The results were obtained over 100 runs.	54
6.3	Table depicting the overhead of tmem functions in terms of CPU cycles when the tmem cache does only local de-duplication. Test used the <i>rand file</i> . <i>get</i> miss overheads were separately calculated without populating the tmem cache. . . . .	55
6.4	Table depicting the overhead of tmem functions in terms of CPU cycles when the tmem cache does only local de-duplication. Test used the <i>one file</i> . <i>get</i> miss overheads were separately calculated without populating the tmem cache. . . . .	56
6.5	Table depicting the overhead of tmem functions in terms of CPU cycles when the tmem cache does both local & remote de-duplication. . . . .	56
6.6	Table depicting the max possible overhead of tmem functions in terms of time in ms when tmem cache does both local & remote de-duplication. The frequency of CPU is assumed to be 1GHz . . . . .	57



# List of Figures

1.1	Effect of ballooning on applications accessing memcached. . . . .	4
1.2	Effect of ballooning on DB query performance. . . . .	5
1.3	Application layer disk I/O throughput in different cache setups for read intensive workloads. . . . .	7
2.1	Tracing cleancache-tmem operations . . . . .	13
3.1	Illustration of out-of-band sharing. . . . .	19
3.2	Illustration of in-band sharing. . . . .	20
3.3	Application Level Ballooning (ALB) architecture. . . . .	22
4.1	A single tmem pool of a VM . . . . .	26
4.2	An illustration of full tmem/VM view . . . . .	26
4.3	An illustration of the hypervisor/system view . . . . .	27
4.4	Tmem backend — VM & System view combined . . . . .	28
5.1	Setting bloom filter bits on a tmem put . . . . .	35
5.2	Bloom filter stats assuming each guest is going to dump its entire max configured memory contents of size 0.5 GB into tmem cache. VMs per host are 16, total co-operating hosts are 16 . . . . .	37
5.3	Bloom filter stats assuming each guest is going to dump its entire max configured memory contents of size 0.4 GB into tmem cache. VMs per host are 16, total co-operating hosts are 16 . . . . .	37
5.4	Tmem backends registering with <i>leader server</i> . . . . .	38
5.5	bloom filter exchange . . . . .	38
5.6	extending local de-dup to use <i>remote sharing candidate list &amp; local only list</i> to support remote de-dup . . . . .	40

5.7	use of <i>remote shared list</i> to keep references to pages that were success- fully remote de-duplicated . . . . .	43
5.8	Illustration of remote de-dup in response to a <i>remotification</i> request . . .	45
5.9	Illustration of a <i>remote get</i> request for a page from a remote backend in which this page was de-duplicated . . . . .	46
5.10	<i>remotification</i> - attempt remote de-dup . . . . .	47
5.11	<i>remotified get</i> - retrieving contents of remote de-duplicated page . . .	47

# Chapter 1

## Introduction

Virtualization tries to maximize the utilization of the underlying hardware resources of a physical machine by hosting multiple virtual machines on it. There have been many efficient solutions for optimizing CPU and I/O device utilization among these VMs. But time-sharing of physical memory is difficult. "Traditional" techniques to efficiently time share memory among VMs involved steps like - measuring current usage and deducing future memory needs based on it, reclaiming memory from those VMs that have an excess memory, elimination of duplicate memory content etc. But there are some fundamental issues related to these approaches owing to the nature of memory as a resource and the way in which memory is consumed by operating systems like Linux.

The CPU is a replenished resource every new second. Memory as a resource is not as "renewable" as the CPU because of the following reasons.

- Before we can reuse memory we need to ensure a lot of consistencies which require a lot of moving things around. This is called *memory inertia*.
- Until recently memory was a scarce, expensive and fixed resource.

In a bid to overcome memory inertia OSs were designed to fill up available memory in a supposedly "intelligent" manner to avoid moving things around. And since memory was available only at a premium, in order to utilize it to the fullest, OSs tried to fill it up as much as it could with what it perceived would be useful content. Thus OSs became notorious memory hogs. More the memory that was given to an OS the more it will consume and try to put to some use. So it is neither easy to predict if an OS really needs more memory nor is it easy to tell whether it is using the available memory efficiently.

Thus in virtualized systems memory management is a hard problem and as a result physical memory is increasingly becoming a bottleneck limiting the efficient consolidation of virtual machines onto physical machines.

## 1.1 Issues with memory usage and management in virtualized environment

Recall that memory, at any given time, can be divided into kernel code & data, application code & data and the page cache. Page cache is a mechanism that is used to overcome slow disk accesses and often it accounts for a major portion of the memory. Page cache is an example of the attempt to maximize the utilization of whatever memory is available by putting it to some good use. If a valid copy of the page being requested is available in the cache then it is a cache hit which avoids a costly disk access and if such a copy is not there then it has to be fetched from the disk. Not only that, the kernel needs to ensure that there is space, in the cache, for the new page being brought in. Hence to avoid the effects of memory inertia the kernel tries to do some sort of prediction as to which pages will be required again in the near future (like *LRU*) and tries to keep only those in the page cache. Now, in spite of this, if there are a lot of misses happening then it implies that the prediction is not working well and that a lot of memory is being wasted in the page cache by holding onto pages that are not being used.

### 1.1.1 Overcommitment & dynamic provisioning of memory

Overcommitment of memory is an unavoidable requirement to ensure improved effectiveness of system virtualization. Overcommitment means that the total memory, configured for all active virtual machines, exceeds the total amount of actual available host memory. When memory is overcommitted as mentioned there should be management mechanisms to dynamically provision it among the VMs. This virtualized memory management method of dynamically provisioning memory among VMs needn't go well together with the native memory management methods in the VM all the time. A classic illustration of this issue is the *double paging* problem related to meta-level page replacement policies used as a virtualized memory management technique in the early days. Double paging occurs when a meta-level policy selects a page to reclaim and pages it out only to find that guest OS, under memory pressure, chooses the very same page to be

written to its own virtual paging device. This causes unnecessary disk accesses to fault in the page contents from the system paging device and to write it out to the virtual paging device.

### 1.1.2 Ballooning & Native memory management

Even dynamic virtualized memory management techniques, like Ballooning, that allow the guest OS to invoke its own native memory management algorithms as a response to its actions needn't fare well together. Ballooning is a dynamic provisioning technique, used by memory management policies, that allows for underutilized physical memory to be reassigned from one VM to another. A pseudo-device driver, called the balloon driver, is made to run in each VM. Rather than communicating with a real device it communicates with the hypervisor. This makes it possible for the hypervisor to get memory from those VMs that it perceives as underutilizing its current allocation. Hypervisor obtains memory from such VMs by making the balloon driver issue `vmalloc` call like a normal device driver operation and transfers the ownership of the pages absorbed in such a manner to itself. The ownership of these pages can then be transferred to the balloon driver in a needy VM and balloon driver can release the pages to this VM. But there is no guarantee that this VM will make use of the memory thus obtained efficiently. But there is graver drawback to this approach. Assume the system is in a state where it has zero memory wastage made possible by ballooning. Now suddenly if there is an urgent need for some memory, maybe to allocate a new VM or to make space for a migrating VM, the system needs to depend on ballooning. But because of memory inertia or the balloon driver being denied the requested memory by the host VM, this urgency can potentially escalate to an out of memory situation. Even if the VM gives up some pages it most certainly going to be from the page cache. This can wreak havoc with the already messed up page cache prediction and cause what is known as *refaults*. A refault occurs when a page which was evicted from the page cache under memory pressure, to reuse the page frame it occupied, has to be fetched back again from the disk.

### 1.1.3 Ballooning & applications managing own memory

In addition Hwang *et al.* (2013) and Salomie *et al.* (2013) have illustrated that ballooning doesn't work well with applications, like memcached, DBs, language runtimes,

that manage their own memory. While applications like memcached store entire key-value pairs in memory only, databases heavily cache data and results to avoid expensive I/O, and language runtimes exploit extra memory to reduce the frequency of expensive garbage collection. Hwang *et al.* (2013) found that as the memory allocated to VM varies over time, pages belonging to memcache need to be swapped to satisfy the balloon driver's request. This causes the response times to rise several seconds and reduces the performance of memcached to a level even lower than when not using a cache at all. Salomie *et al.* (2013) has showed us that ballooning causes the memory being used by these applications to be transparently paged to disk disturbing their performance optimization measures. Thus a combination of native memory management technique like page cache or applications that manage their own memory doesn't go well together with virtualized memory management technique, like ballooning. A combination of any one from the former and the latter can be caught up in a vicious circle.

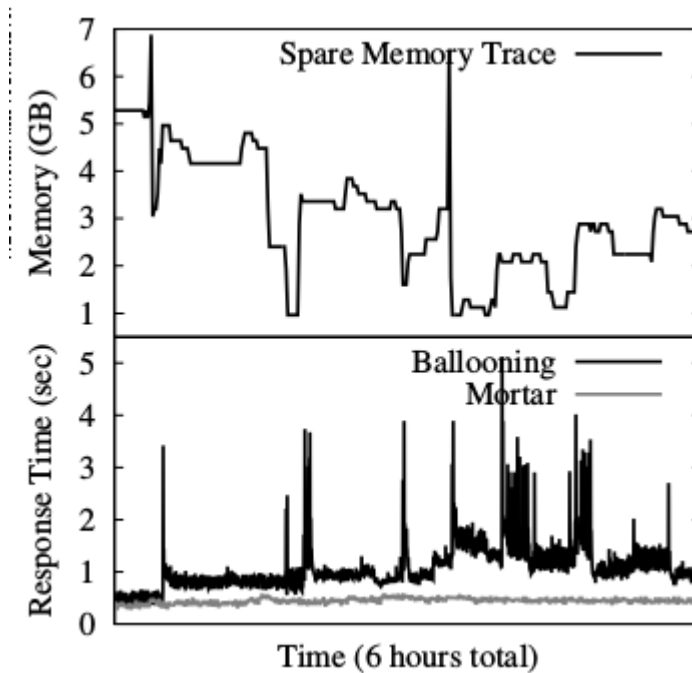


Figure 1.1: Effect of ballooning on applications accessing memcached.

Source: (Hwang *et al.*, 2013)

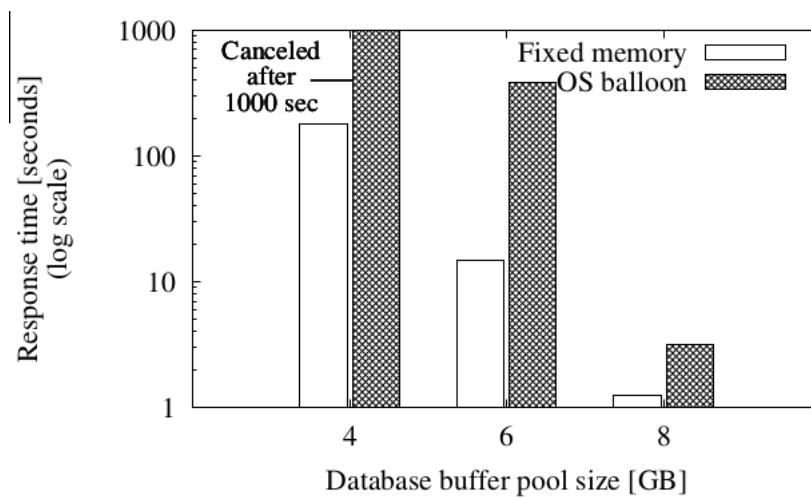


Figure 1.2: Effect of ballooning on DB query performance.

Source: (Salomie *et al.*, 2013)

## 1.2 Hypervisor Caching: Another memory management technique

From previous sections we've seen that virtualized memory management techniques like working set estimation, prediction of memory requirement and re-assignment using ballooning will never be sufficiently accurate. This inaccuracy becomes glaring when it combines with native memory utilization techniques like page cache. Hence we're in the dark when it comes to deciding how much to recover at what rate from a VM because of this inaccuracy. Hypervisor caches, a relatively recent paradigm in virtualized memory management, provides a mechanism for mitigating the impact of that inaccuracy by acting complementary to dynamic provisioning techniques like ballooning. Basic idea of hypervisor cache is to provide a pool of memory in hypervisor, controlled by the hypervisor and to be accessed only via the hypervisor, for the benefit of VMs in a way that minimizes impact of short term mispredictions and memory inertia, which are made worse by the memory pressure caused by dynamic partitioning.

Hypervisor caches, like **Transcendent memory** (tmem), allows the VMs to *put* and *get* objects into/from the hypervisor cache's memory pool. Guest OSs that are accessing the tmem pool are called **clients**. The hypervisor controls the state of the client objects in tmem and can vary the size of the tmem pool based on its memory management policy.

Such a hypervisor cache is a good candidate to become a second chance cache. If tmem acts as a second chance cache, clients (i.e. VMs) under memory pressure can make use of this volatile memory to store its evicted page-cache pages belonging to different files (*client objects*) accessed by VM. This can mitigate the effects of refaults by reducing disk access latencies in case these evicted pages are again demand in the VM. Thus tmem can be thought of as a second chance exclusive cache. Tmem can provide ephemeral or persistent semantics for holding onto the client objects inserted into the cache. Tmem also has the option to employ de-duplication, compression of the contents held by it to enhance the utilization of memory under it. And when the hypervisor itself is under memory pressure, because of overcommitment or other reasons, it can evict from the tmem pool and reclaim memory to ease the pressure. Thus it is evident that such a hypervisor cache along with the techniques like memory sharing, compression can act complementary to ballooning by mitigating the effect of refaults. And together these have the potential to become a much better memory management policy.

### 1.3 Motivation

The effectiveness of a setup consisting of guest page cache (G) combined with an exclusive second chance page cache (GT) in the hypervisor that does compression has been been illustrated by Mishra and Kulkarni (2014). That work compares this setup with other possible cache provisioning options. Results there have shown that for read intensive workloads this setup has more than 6x disk I/O throughput, upto 20x lower disk read size when compared to the default inclusive caching combination i.e. guest page cache (G) with host page cache (H).

Performance metric	Performance
Disk I/O throughput	6x more
Disk read size	20x lower

Table 1.1: Effectiveness of G-GT compared to G-H for read intensive workloads



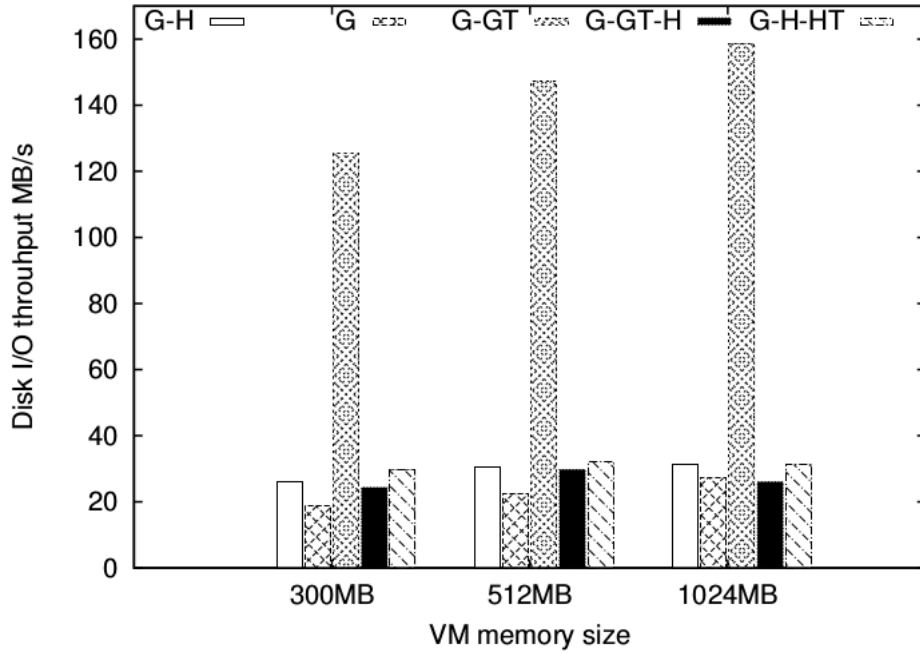


Figure 1.3: Application layer disk I/O throughput in different cache setups for read intensive workloads.

Source:(Mishra and Kulkarni, 2014)

In most data centre setups VMs access the same secondary store over the network. There is a clear benefit in avoiding refaults or disk access in general here. Moreover data centres are bound to have a lot of content similarities owing to — the similarity in applications running in the VMs, common files being accessed, booting VMs from same templates/base images. Previous research have determined that out of all the shareable pages of co-hosted VMs between 64% - 93% are part of the page cache (Kloster *et al.*, 2007), (Miłós *et al.*, 2009). Taking into consideration the above points and the performance benefits of a second chance page cache in hypervisor, we found an opportunity to introduce one that tries to improve the utilization of the memory available to it by looking to eliminate duplicate memory content via exploring sharing opportunities in other physical machines as well. This is under assumption that a disk access over a network would be slower than a RAM access over the same.

Mortar tried to pool together the spare memory in the physical machines of a datacentre and tried to expose it as a volatile data cache to the applications, like memcached, running in VMs hosted on those physical machines. They illustrated that an approach like this can help in mitigating the negative impact ballooning has on applications that manage their

own memory. But we found out that they were merely letting the application's distributed nature to actually access the remote memory and utilize the sharing opportunities. This solution is limited to and tightly coupled to the memcached application that is inherently distributed by nature. This solution also has the overhead of traversing the application-kernel-hypervisor boundaries multiple times to enable remote sharing. Thus we believe it missed an opportunity to explore the full potential this scenario is presenting us.

## **1.4 Problem Statement**

1. Design and implementation of a hypervisor cache, as a second chance exclusive page cache, that is capable of doing de-duplication of the content being held in it in a distributed manner when under memory pressure i.e to explore and pursue page sharing opportunities among different instances of the same hypervisor cache in different physical machines. Thereby improving the utilization of the memory available to it and improving application disk I/O throughput by reducing costly disk accesses.
2. Demonstrate the correctness of implementation.
3. Evaluate the overheads of implementation using micro level benchmarks.

## **1.5 Outline of the Report**

The rest of this report is organised follows. Chapter 2: Background, Chapter 3: Related Work, Chapter 4: Design, Chapter 5: Evaluation of implementaion, Chapter 5: Future work & conclusions.

# Chapter 2

## Background

This section delves into the background required to understand the working of transcendent memory, the hypervisor cache being explored and extended in this thesis. This includes understanding its design & API, the existing use cases of tmem in Linux and a few implementations.

### 2.1 Transcendent memory basics

Earlier sections made it clear that we make use of transcendent memory to store and retrieve objects of different clients (i.e.VMs). So the tmem store is to be divided among all the VMs that are accessing it. Therefore tmem provides an API to the VMs to create/destroy their own memory pools, put and get objects in it using a key, invalidate the contents of an object for a given key. This API is a part of the backend interface to tmem. These APIs are to be invoked via a `hypercall` which will deliver parameters required for tmem operation to the tmem implementation. Hence tmem API is paravirtualized.

#### 2.1.1 Tmem pools

Each tmem client can request to create a new pool and delete the pool using APIs provided by tmem. Clients have to create atleast one pool to put their objects in tmem. Pools of the same client can be distinguished by its unique pool id. There can be different semantics associated with a client's pool. E.g. Shared, Private, Ephemeral or Persistent. These semantics characterise the life span and visibility of objects in the pool. A client

can specify the semantics to be associated with any of its pools during pool creation with the help of flags passed to the API.

Table 2.1: Different types of tmem pools.

<i>flags</i>	<b>Ephemeral</b>	<b>Persistent</b>
<b>Private</b>	Private to each VM. But memory can be reclaimed from this any moment. Hence pages successfully put to an ephemeral pool may or may not be present later when the client kernel uses a subsequent get with a matching handle.	Private to each VM. Pages successfully put to a persistent pool are guaranteed to be present for a subsequent get.
<b>Shared</b>	Shared among VMs co-located on a physical server that share a cluster filesystem. Since the filesystem is shared, shared tmem pool is most appropriate as only one page frame is to be used in tmem to store a page even if multiple VMs are accessing it.	Shared-Persistent pools have not been explored much.

### 2.1.2 Tmem API

Since objects in tmem cache are files as discussed in section 1.2 the key which uniquely identifies it should be associated to a unique file identifier in the VM. In Linux such an unique file identifier is the `inode` number. Since inode numbers are unique only within a partition/disk it would be a natural choice to put objects from the same partition/disk into the same pool. But it is not necessary that there should be a one-to-one correspondence between a tmem pool and a partition/disk. Either ways it is now clear that a client can have multiple pools of memory within tmem and pool identifiers to distinguish between them. In addition a tmem operation should have the `offset` index of the page within the file block which is being put/get into tmem and finally a reference to the actual VM page containing the data. The `<poolid, objectid, index>` tuple is referred to as a handle. The following are the tmem interface specification as per the available spec.

`TMEM_NEW_POOL(UUID, flags)`: Create a new pool in transcendent memory.

`TMEM_DESTROY_POOL(pool_id)`: Destroy a tmem pool and free all data.

**TMEM\_PUT\_PAGE(pool\_id, object\_id, page\_id, pfn):** Copy a page of data from a physical page frame into a tmem pool and associate it with a handle-tuple.

**TMEM\_GET\_PAGE(pool\_id, object\_id, page\_id, pfn):** Lookup a page of data in tmem associated with a handle-tuple and, if found, copy it to a physical page frame.

**TMEM\_FLUSH\_PAGE(pool\_id, object\_id, page\_id):** Disassociate a handle-tuple from any data in tmem.

**TMEM\_FLUSH\_OBJECT(pool\_id, object\_id):** Disassociate all pages associated with an object from any data in tmem.

## 2.2 Tmem use cases

### 2.2.1 Frontends

There can be different sources for the data to be stored in the tmem backend pools. These sources of data are called tmem frontends and are the best illustrations for tmem use cases. The occurrence of re-faults and associated costly disk accesses can be brought down if the page cache pages being evicted are held in tmem. Similarly disk access can be avoided if the swapped out anonymous pages are stored persistently in tmem till those are requested back.

Table 2.2: Advantages of tmem to different use cases

Related to page cache pages	Related to anonymous pages
Re-faults are avoided thereby reducing the associated disk read latency for read intensive workloads.	Swapped out pages are kept in memory thereby avoiding the associated disk write and read when these pages are requested back.

The use case related to caching evicted page cache pages is called *cleancache* and the use case associated to caching swapped out anonymous memory is called *frontswap*.

### 2.2.2 Cleancache

Cleancache is a tmem frontend interface, resident in the guest OS, using which VMs can access the tmem backend API introduced in 2.1.2. Cleancache provides generic API and its invocations are translated by a shim layer in guest OS to appropriate hypercalls which will transfer the arguments to the hypervisor. In the hypervisor another shim layer

parses these arguments and invoke the appropriate tmem API.

Cleancache is the source of data for tmem that acts as an exclusive second chance page cache in the hypervisor. Hence cleancache is closely associated with the page cache and the VFS in guest OS. Cleancache hooks are placed at appropriate places in the *filesystem* layer. When the guest OS reclaims the page frame of a clean page cache page the cleancache hooks channel the page's contents via the guest OS shim layer, the hypercall and translates it into a put in the tmem pool. Since cleancache provides clean page cache pages the tmem pool associated with cleancache chooses **ephemeral** semantics. This means that the pages stored in it may be discarded, if tmem chooses to, without being concerned about writing it back to disk. Later, when the guest OS needs that page, it requests it back from tmem via a cleancache get call which gets translated to a tmem get. If the page was retained by tmem it is given back and removed from tmem; else the guest OS needs to proceed with the refault, fetching the data from the disk as usual. Cleancache is also responsible for ensuring coherency among the page cache, tmem and disk for parts of the same file held in them. So cleancache hooks to flush the corresponding tmem data are to be inserted where ever the guest OS might invalidate the data e.g when a part of the file is deleted. Figure 2.1 illustrates the flow of a tmem operation starting from a file read issued by an application running in the VM. Each filesystem needs to explicitly choose to use cleancache when it is mounted.

### 2.2.3 Frontswap

Frontswap is another tmem frontend interface in the guest OS. Frontswap is the source of data for tmem that exists as an in memory swap "disk" at the hypervisor. Thus frontswap allows the Linux swap subsystem to use tmem as a swap space. When the guest OS, under memory pressure, frees up page frames containing anonymous memory the contents of those pages should be to be swapped out to a swap device. Historically swap devices have been disk based. Disk based swap devices are several orders of magnitude slower than RAM, so swapping has significant performance issues. Section 1.1.3 had illustrated the problems applications that manage own memory had with ballooning. Since these applications also deal with lot of non-file backed anonymous memory the combination of frontswap and tmem as an in memory swap disk can really help in mitigating effects of those problems. As the memory available to tmem is dynamic it is not necessary that the guest OS is always going to successfully find space in tmem to swap

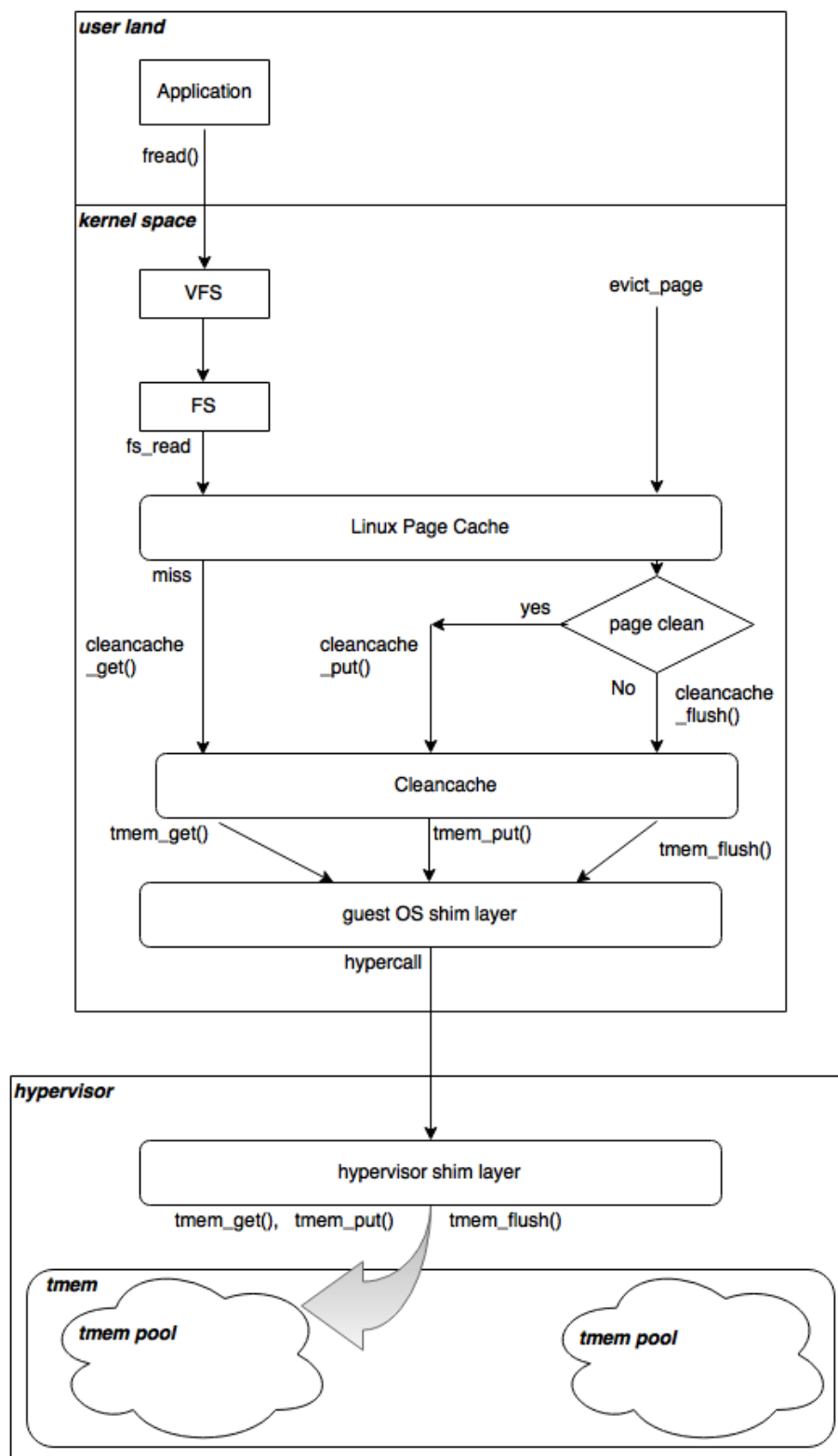


Figure 2.1: Tracing cleancache-tmem operations

out pages. Whenever a page needs to be swapped out, if there is space with the tmem, the swap subsystem can successfully swap out to it. If tmem rejects this request then the swap subsystem needs to proceed to the swap device as usual. If the request was a success then pages swapped out to tmem are guaranteed to be retrievable at any time later. Thus tmem backend for frontswap has persistence semantics. The client swap subsystem can flush the swapped out pages from tmem when it is certain the data is no longer valid.

## 2.3 Tmem backends

Tmem can be made to store data from the frontends in different ways. These implementations of tmem are known as tmem backends. Each tmem backend will have its own way of organising, managing data stored in it. Its upto the tmem backend developer to choose the features and attributes. E.g. whether to support creation of both persistent & ephemeral pools or to support only one of those etc. Theoretically one hypervisor can have different backend implementation existing at the same time. But all existing works have concentrated on having just one backend in a hypervisor which may be configured for any number of frontends. This thesis also follows suit.

A tmem backend can choose to compress data held by it, thereby increasing the utilization of backend capacity; do content based de-duplication of the pages held by it, again resulting in an increased utilization of backend capacity; distribute the backend store across co-operating instances of the same backend in other hypervisors, thereby balancing the load on a cluster of participating hosts; pursue de-duplication opportunities in other instances of the same backend in other hypervisors, to increase the utilization of backend capacity. The following subsections introduce a few existing tmem backend implementations.

### 2.3.1 Xen tmem backend

Tmem was originally conceived for xen and so the xen implementation is the most mature. The tmem backend in xen utilizes spare hypervisor memory to store memory pages from both cleancache and frontswap guest frontends, it supports a large number of guests, and optionally implements both compression and deduplication to maximize the amount of data that can be stored in it. The tmem frontends calls are converted to xen hypercalls using a shim. Xen tmem also supports shared ephemeral pools. So guests co-located on a physical server that share a cluster filesystem need only keep one copy



of a cleancache page in tmem. Xen tmem is also equipped to support live migration and save/restore of tmem clients. In order to avoid any single client from denying tmem services to other clients, provisions to apply limits or weights on tmem backend usage of each client is available in Xen tmem implementation.(Article, 2010)

### 2.3.2 Zcache

Zcache is a tmem backend that was envisioned for non-virtualized environment. This in-kernel tmem backend does compression of the data that it holds. By using compression zcache can substantially increase the number of pages it stores in RAM.

Though zcache implementation is currently unavailable in linux mainline tree, it was available as a staging driver in previous releases. It handles both persistent pages, from frontswap, and ephemeral pages, from cleancache. It uses in-kernel routines to compress/decompress the data contained in the pages. The essence of zcache is in finding free page frame under its allocation and finding a pair of compressed pages, “compression buddies”, that will fill up this free page with least internal fragmentation. Since tmem can reject any page when a put is attempted, data that compress poorly can be turned away. This poorly compressible data can be discarded if its source was cleancache or can be stored in the original swap disk, if it originated from frontswap. In response to a get from the frontend, zcache needs to identify the page frame holding the compressed page corresponding to the request, decompress the page to the location specified by the get request. It is clear that zcache is trying to use the idle CPU cycles to execute compression and decompression algorithms so that more data can be kept in RAM. Hence there is a possibility of cpu-memory cost-benefit trade-off. (Article, 2013)

### 2.3.3 RAMster

RAMster is a tmem backend that does both compression and remote storage; compression to increase utilization of tmem and remote storage to reduce memory load on the system because of tmem. Similar to zcache, RAMster was also envisioned for non-virtualized environment. RAMster can be defined as a peer-to-peer tmem implementation where a “cluster” of kernels dynamically pool their RAM. This enables a RAM heavy workload on one machine to temporarily and transparently make use of the RAM on another machine which is presumably idle or has a relatively lighter workload. This remote access of RAM happens in an asynchronous way. When a page is put to RAMster, it is

compressed and stored locally. Periodically, a thread will **remotify** these pages by sending them via messages to a remote machine which is willing to take it up. The local kernel keeps track of where the remote data resides. Later when a **get** request is issued for a page from the RAMster backend, if the data is not locally available, a message is sent to fetch the remote data and the requesting thread is made to sleep. Once the remote data fetch is complete the data is decompressed, requesting thread is woken and the **get** request is serviced. As RAMster also has the idea of compressing its contents it can be seamlessly combined with zcache. Such an implementation would distribute compressed cleancache or frontswap pages among peers to reduce memory load induced by zcache in the kernel. In fact a zcache-RAMster combined implementation was available in the Linux mainline a few releases back.(Oracle, 2010)

#### 2.3.4 KVM tmem backend

KVM is a hosted hypervisor implemented as a module in the native stand-alone Linux. Implementing tmem for KVM, therefore, is no more arduous than coding in the Linux kernel. It can be implemented like any other device driver. In fact zcache and RAMster, as mentioned in 2.3.2 and 2.3.3, are implemented as staging device drivers. The only specificity required to figure 2.1 to channel cleancache and frontswap frontend calls to KVM tmem would be to use KVM specific hypercalls.

Because of the above mentioned ease of implementation and the fact that mainline Linux kernel doesn't have a KVM tmem implementation this thesis chose to implement the proposed tmem backend, one that does de-duplication based on local and remote content, in KVM.

# Chapter 3

## Related Work

Techniques for managing virtualized memory can be broadly classified into following types — (a) Ballooning based dynamic memory provisioning techniques and (b) Content similarity elimination techniques (Mishra and Kulkarni, 2014). This section briefly presents the basic idea behind past works belonging to this genre even though hypervisor caches can be classified as neither of these. It would be helpful to delve about it as hypervisor caches are meant to complement the existing management policies based on these techniques.

### 3.1 Determining memory usage statistics & finding an optimal balloon size

The effectiveness of a dynamic memory management technique depends on how accurately it can determine the memory usage/requirement statistics of a VM. There is an option of obtaining the memory usage statistics from information maintained in the guest OS. This may not be an accurate determination as various guest OSs may be using different metrics for their individual estimations. Working set size or WSS is often taken as a measure of memory usage. WSS is the amount of physical memory that is being actively used by an application out of its total allocation in a recently concluded epoch of time. Hence WSS estimation at well thought out regular intervals can give us an idea about how much memory a VM needs or can give up. If the WSS is smaller than the actual memory assignment to the VM, it can give up some of the unused memory without affecting its performance. Waldspurger (2002) discusses a statistical sampling approach to obtain the

WSS of each VM as a whole. A sort of exponential weighted moving average is used to smooth out estimates across sampling periods. They then extrapolate the usage statistics to obtain the statistics for the entire system. Zhao *et al.* (2009) uses a Page Miss Ratio Curve or MRC based WSS estimation. This is a more accurate measure of the memory utilization for a given allocation rather than the sampling based approach. MRC is obtained from LRU histogram and the basic idea behind it is the Mattsons Stack Algorithm explained in Zhou *et al.* (2004). This algorithm gives us the page miss rates possible for different amounts of memory allocation using the recent memory access patterns. Thus by looking at this information the VMM can deduce the memory allocation to be made to a VM to achieve a more tolerable page miss rate. Thus the common thread throughout all these approaches is an attempt to find out, as accurately as possible, the current memory usage and demands of a VM; and using this information to drive memory allocation/reclamation by choosing an optimal size for the balloon driver in the VM.

### 3.2 Content based de-duplication

Since content based de-duplication is a major thread of this thesis this section devotes some time on this memory management strategy. Studies have pointed out the high presence of duplicate content among VMs co-hosted in a physical machine (Kloster *et al.*, 2007). In section 1.3 the reasons for high presence of similar content was pointed out. There are mainly a couple of choices available while doing de-duplication — (a) the granularity at which similar content is to be eliminated, (b) when to search for similar content. Based on decisions arrived at for these choices many de-duplication policies can be formulated.

If the granularity at which de-duplication is pursued is very fine, like at byte level or data structure level, the amount of memory retrieved can be maximized. But such a strategy will not only incur overheads in processing (e.g. searching for duplicates) but also in maintaining and managing the information related to large number of similar contents. If the granularity is larger, say page level, these overheads can be avoided but at the cost of missing chances to de-duplicate lot of smaller sized similar contents. Another choice is the “when” to look for duplicates. Choosing to look for duplicates out-of-band or asynchronously has associated scanning overheads. Having an aggressive scan rate helps in finding even short term sharing opportunities, of course at the expense of costly CPU cycles. On the other hand an in-band or synchronous pursuit of de-duplication in the disk

access path can avoid processing overheads significantly. This is due the fact the main memory is empty at boot time. The drawback of this approach is that this misses out on short term sharing opportunities presented by anonymous regions. But if the presumption that anonymous memory allocated by processes and kernel have lower lifespan and are prone to frequent changes than disk backed pages holds good then this approach is justified.

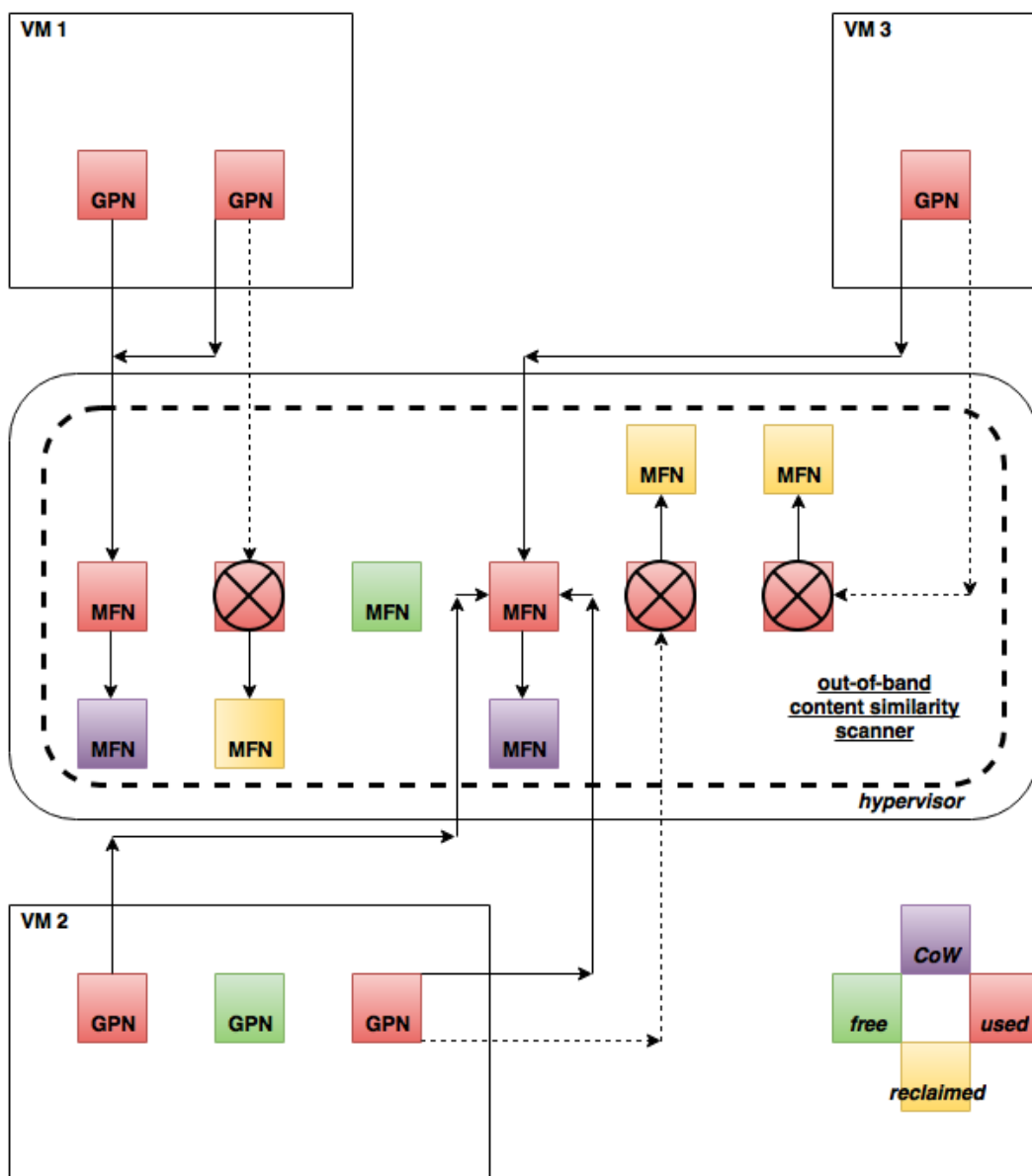


Figure 3.1: Illustration of out-of-band sharing.

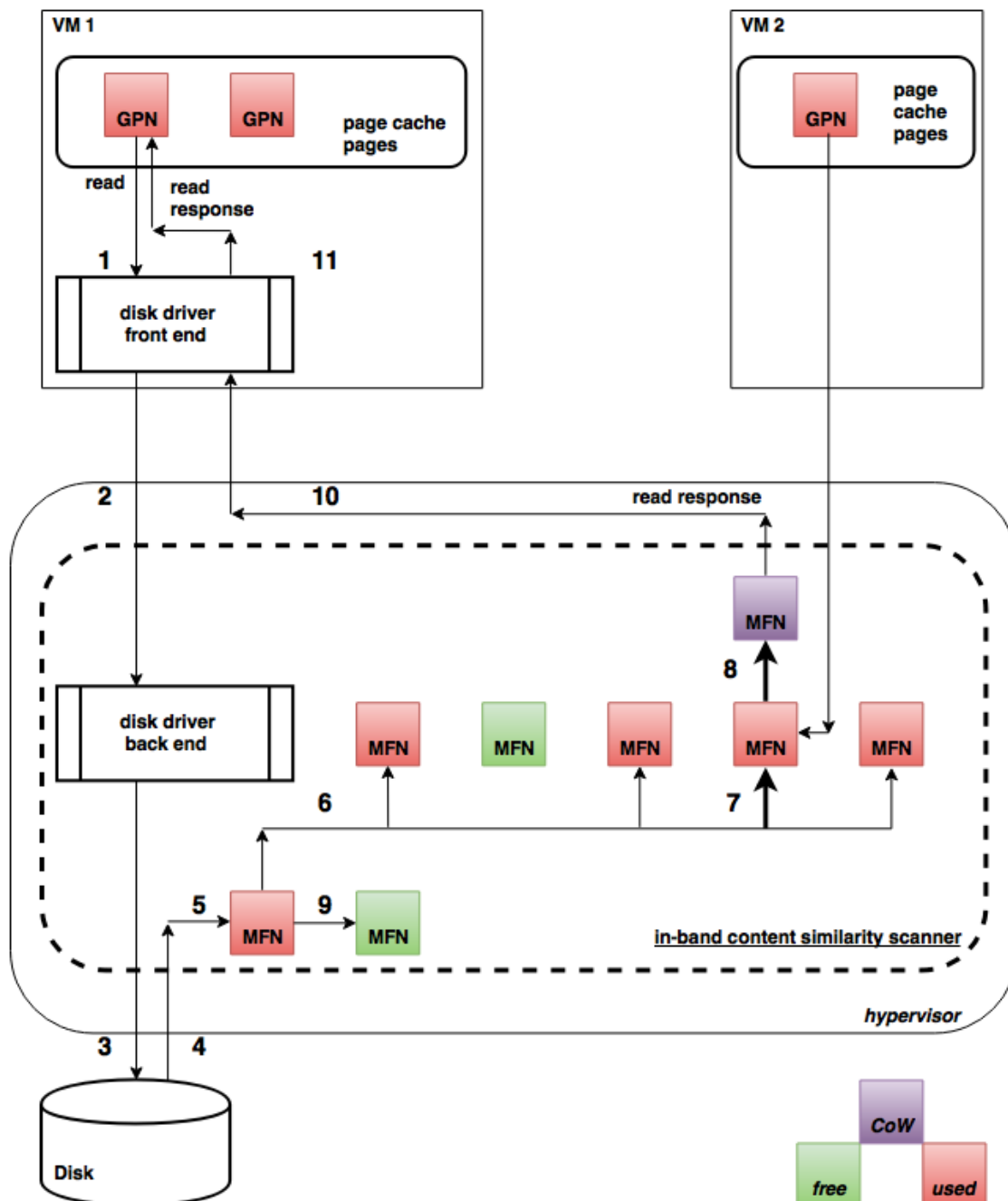


Figure 3.2: Illustration of in-band sharing.

CoW — copy-on-write is the technique used to eliminate the  $(n-1)$  duplicate copies and keep just 1. When a VM tries to modify a host page frame that is marked CoW a VMEXIT is triggered; the hypervisor breaks the sharing the guest page had with this host page frame, allocates a new host page frame and changes the guest page to host page

frame mapping. Breaking of CoW has associated CPU processing costs.

The survey paper (Mishra and Kulkarni, 2015) has summarized most of the aspects of content based de-duplication, the choices available, its drawbacks and also briefs about past work done in this domain. Waldspurger (2002) identifies pages with the same content by out-of-band scanning within the hypervisor which does not involve any kind of modifications to the guest OS. Hashing is used to identify the possible matches as a first step. In case the hashes match, a full comparison of the pages are done. CoW technique is used to share the pages. If an attempt is made to write to the shared page, a private copy is created for the writer. Rachamalla *et al.* (2013) focuses on an out-of-band sharing technique, KSM (Kernel Samepage Merging), used by the Linux hosted hypervisor KVM. This work compares the sharing achieved by KSM against the total sharing opportunities presented by VMs in various settings. It provides insights into the various KSM configurations, their impact on resource utilization and application performance characteristics. The work also quantified the cost-benefit trade-off of KSM in terms of the savings achieved by KSM and associated costs for workloads exhibiting different sharing opportunities and memory usage. They concluded that KSM needs to have an adaptively configurable aggressiveness setting which adapts to changes in memory usage and to the total memory available for sharing. Miller *et al.* (2013) proposed to combine out-of-band and in-band duplicate searching strategies to give priority to new sharing opportunities and to exploit short term sharing chances. They proposed an initial scan of the whole memory once the system boots up and from then on giving priority to memory pages involved in disk accesses. Miłós *et al.* (2009) implements an in-band duplication detection mechanism using what they call “*sharing-aware block devices*”. They emphasize and concentrate on the sharing opportunities that originate from within disk page cache at the time that cache is populated. They also re-distribute the pages reclaimed via de-duplication to the VMs in proportion to the amount of memory being shared by each VM. They penalize the VMs for breaking a CoW sharing by making the “culprit” VM to give up another of its guest pages so that the associated host page frame can be used for the private copy that the VM needs. The VM maintains a list of relatively unimportant, unshared pages as a **repayment FIFO** towards this purpose.

### 3.3 Virtualized memory management for applications managing own memory

Salomie *et al.* (2013) explores an application driven memory management strategy that makes use of a modified ballooning mechanism to vary the applications memory allocation, enhances collocation of VMs by efficiently consolidating available memory and which also preserves the applications ability to optimize its own performance based on having an accurate idea of available memory.

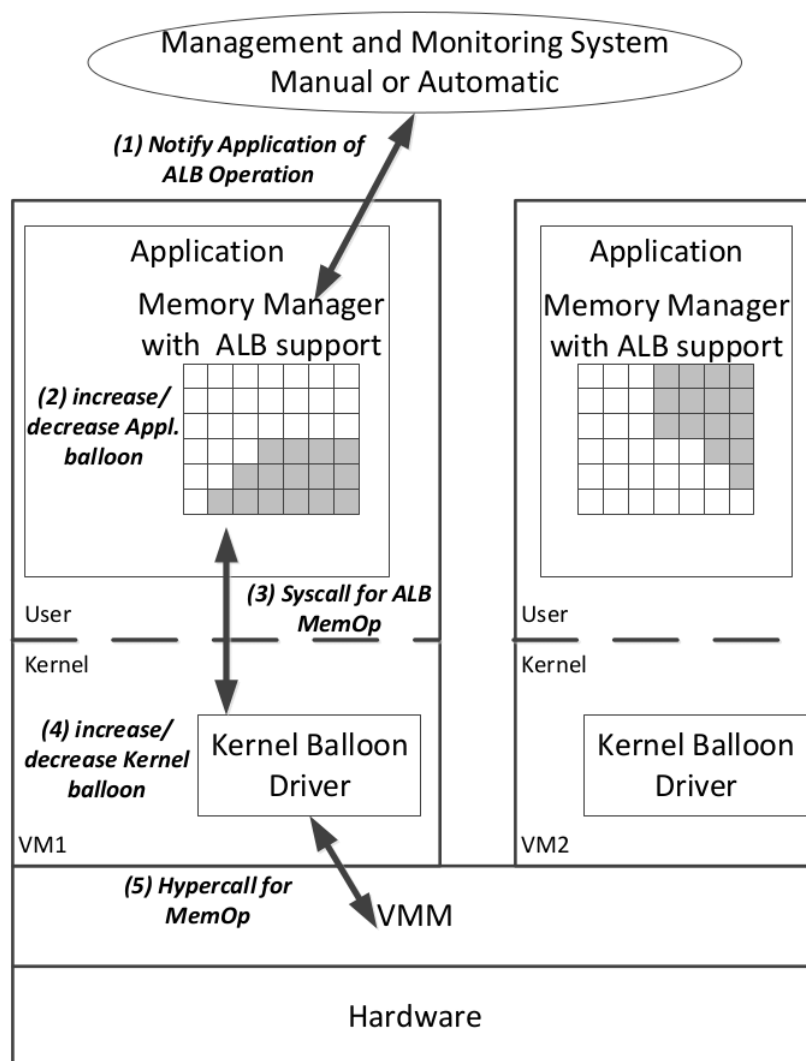


Figure 3.3: Application Level Ballooning (ALB) architecture.

**Source:** (Salomie *et al.*, 2013)

They implemented Application Level Ballooning or ALB for MySQL and OpenJDK.



ALB extends MySQL to support ballooning to and from its buffer pool. While OpenJDK is extended to support the same using its heap. Rather than letting the guest OS reclaim pages of these applications as a response to kernel balloon driver inflation ALB enables these applications to specify those pages that it can afford to loose to the kernel balloon inflation. This is achieved via the following steps: (a) An ALB management system announces the number of pages that the application has to give up - the ballooning target for the application. (b) The application chooses which pages to let go off. (c) Notifies the OS of these pages via a system call. (d) The modified kernel balloon driver will process those. (e) The balloon driver informs the hypervisor of the availability of these pages for its use via a hypercall. Once such pages are given up for reclamation by ballooning those will not be used by the application. MySQL supplies the pages required for balloon inflation from its backend fixed size buffer pool which it creates during startup. OpenJDK supplies these pages from the JVM heap space that is un-allocated and also from heap space associated to recent allocations. Since this mechanism deals with the heap, ballooning operation is tightly coupled with *garbage collection* to an extend that a ballooning operation triggers a garbage collection.

### 3.4 Hypervisor managed cache

Since hypervisor caches are the main theme of this thesis they have been discussed in detail in sections 1.2, 1.3, and 2.3.

# Chapter 4

## Design — local de-duplication

Chapter 3 made it clear that previous implementations of transcendent memory missed out on one or more of the following optimizations or improvements possible to hypervisor caching: (a) of having local and/or global optimizations based on elimination of similar content. (b) of avoiding overheads of frequent user, kernel, hypervisor transitions. (c) of having a generic, application agnostic solution.

### 4.1 Design steps

Before the design is explored a brief note on the design inspirations. The design of a generic, application agnostic, in-hypervisor, second chance page cache that employs local and global optimization based on content similarity will involve the following steps:

- (1) designing a part that does content based de-duplication among all page cache pages put in the cache by different VMs collocated on the same physical machine.
- (2) extending the above design (4.1) to make it co-operate with peers by looking for similar content in the peers' cache and use the similarities found to do de-duplication so as to free up some local memory.

### 4.2 Design inspirations

Section 2.3.4 had pointed out the rationale behind choosing KVM to implement such a `tmem` backend. The options available on the design table were to either go for a from scratch design or to port the design of a similar `tmem` implementation to KVM and make the necessary changes. Since Xen has a full fledged, mature `tmem` backend that accomplished

local de-duplication it was decided to adopt their design and port their implementation to KVM.

Fan *et al.* (2000) illustrated the working of a new approach to *web cache sharing* called Summary Cache. This thesis adopts their idea of using bloom filters to represent the summaries of the contents held by peers. This helps in reducing network traffic among peers by avoiding multicast queries for finding duplicates.

### 4.3 Tmem backend with local de-duplication — design & implementation

This section gives a brief overview of how Xen organises the contents in its backend and how it achieves de-duplication. Since the tmem backend is divided among VMs, each VM has its own view of the backend and the objects that it put there. In addition there is a *hypervisor's view* or *system's view* of all the pages that are available in the tmem backend. While the VM specific view enables a VM to do easy retrieval of the pages belonging to it from the backend, a hypervisor view helps in looking for local as well as remote duplicates without worrying about a pages's ownership.

#### 4.3.1 The guests' view of tmem

##### 4.3.1.1 Guest pool

The VM's view in Xen tmem backend begins with a tmem pool private to this VM (we are not concerned with shared tmem pools in this thesis). Recall that there can be more than one tmem pool for a VM (2.1.2). The objects that are put in a VM's pool are arranged as an `rbtree`. This is achieved by embedding an `rbtree_node` reference within the object. Instead of having a single `rbtree` the pool is organised as a collection of such `rbtrees` in order to prevent the single `rbtree` from growing too big (if the tree becomes too large search time increases). A fixed length hash of `object_id` address is the key to access an `rbtree_node` within the `rbtree` and to get to the object embedding it. The roots of these `rbtrees` are available in an array and can be accessed based on the first byte of an object's address. Thus there are 256 slots in this array and 256 `rbtrees` possible. This arrangement can be thought of as an hash table where the collisions are resolved using `rbtrees`. Refer figure 4.1.

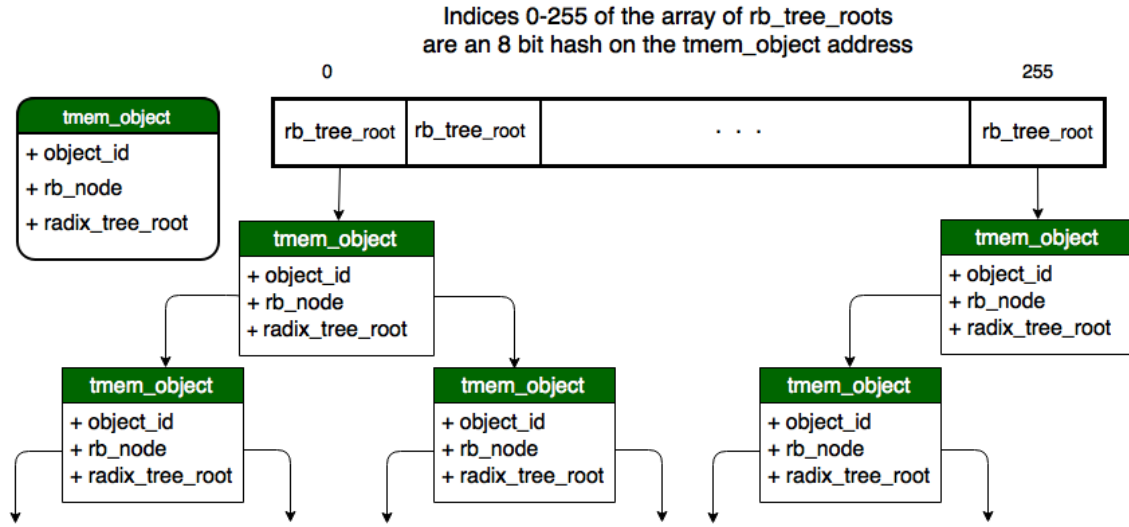


Figure 4.1: A single tmem pool of a VM

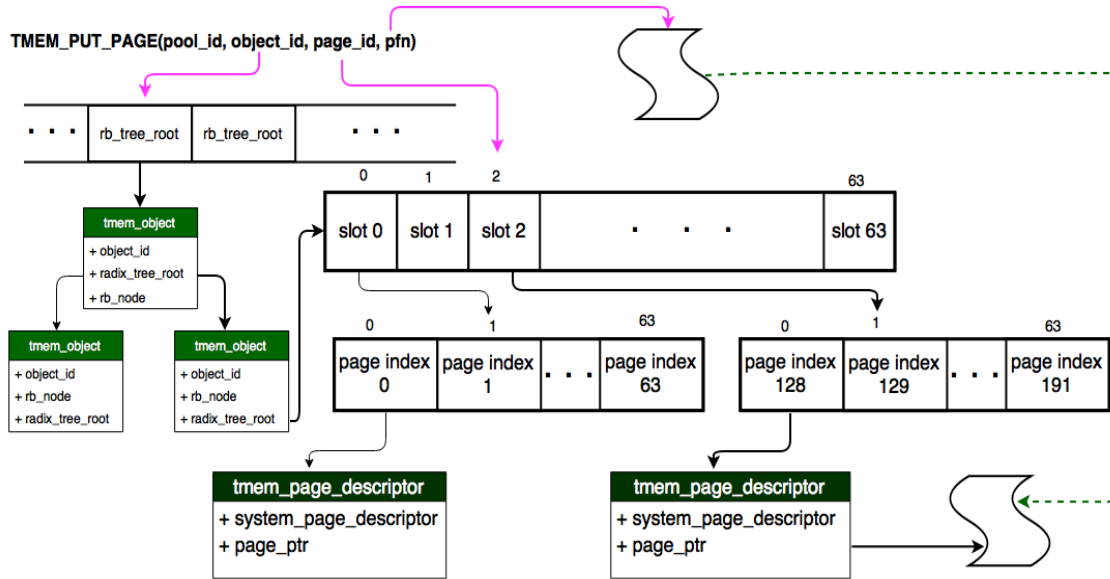


Figure 4.2: An illustration of full tmem/VM view

#### 4.3.1.2 Guest objects

Each node in an `rbtree` mentioned in 4.3.1.1 can refer to an object because the `rbtree_node` is embedded in an object. The object id is an `inode` number because it corresponds to a file. Embedded within each object is a `radix tree` also whose leaves point to structures embedding pages belonging to that object i.e. the pages of an object are organised as a

radix tree. Individual pages in the radix tree are accessed using page numbers. Refer figure 4.2. **Note:** The pool id, object id and the index within the object are obtained from the tmem handle-tuple present in a tmem API call as mentioned in section 2.1.2.

### 4.3.2 The hypervisor's view of tmem

For the hypervisor the pages belonging to an object of a VM are nothing more than machine frames. The hypervisor has no need for distinguishing a page allocated for tmem on the basis of VMs. Hence to keep track of all the pages allocated for a tmem backend there is a separate system or hypervisor view that is devoid of any VM specific tmem notions like pools, objects, tmem pages etc. This view consists of a structure that embeds - (a) a reference to a system/hypervisor page, (b) a reference to an `rbtree_node`; and the `rbtree` that points to the `rbtree_nodes` embedded within the structure just mentioned. As the hypervisor has no other distinguishing attributes for the structure that embeds system/hypervisor pages, the key to the `rbtree_nodes` within this `rbtree` are the contents of the pages itself. A similar approach to section 4.3.1.1 is taken to prevent the single `rbtree` here from becoming too big. Here also there is a collection of `rbtrees` whose roots are accessible from an array of `rbtree roots`. This array can be accessed using the first byte of a page's content. Hence it is clear that we can have 256 such `rbtrees`. Refer figure 4.3.

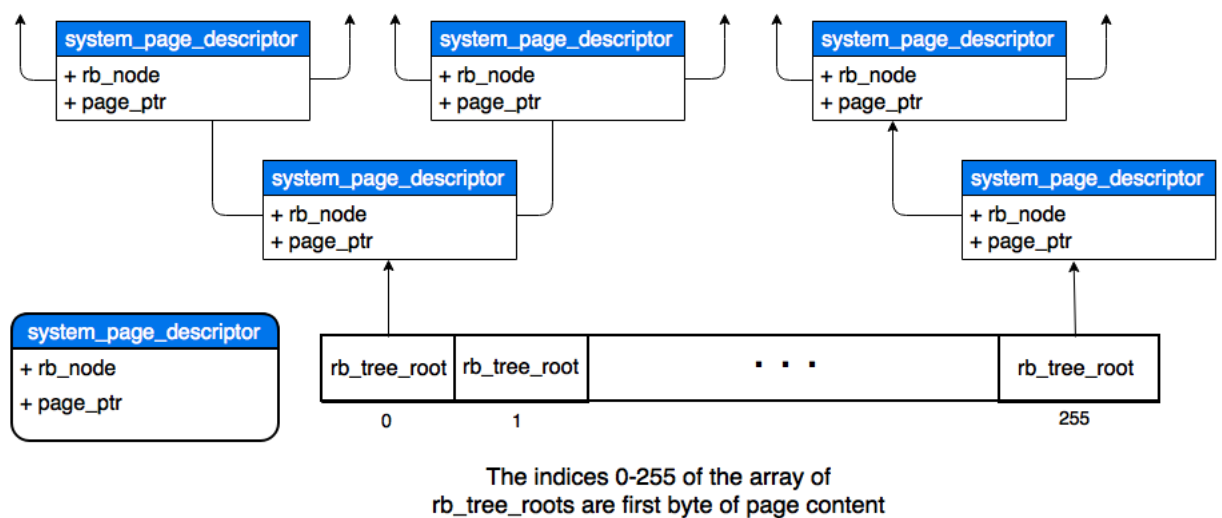


Figure 4.3: An illustration of the hypervisor/system view

### 4.3.3 Linking guest and hypervisor tmem views

Section 4.3.1.2 showed that the leaves of the radix tree embedded within each tmem object point to structures that embed pages belonging the same tmem object and section 4.3.2 showed that the rbtree in system/hypervisor view points to structures that embed pages available in tmem backend. From this point on the structure that embeds pages belonging to a tmem object will be referred to as a **tmem page descriptor** and the structure that embeds pages belonging to a system's view as **system page descriptor**. It should be evident by now that ultimately a single page in tmem should be pointed to by both tmem page descriptor and system page descriptor. This is to ensure that any page put in tmem is a valid page associated to a VM and is accounted for by the system view. Refer to figure 4.4

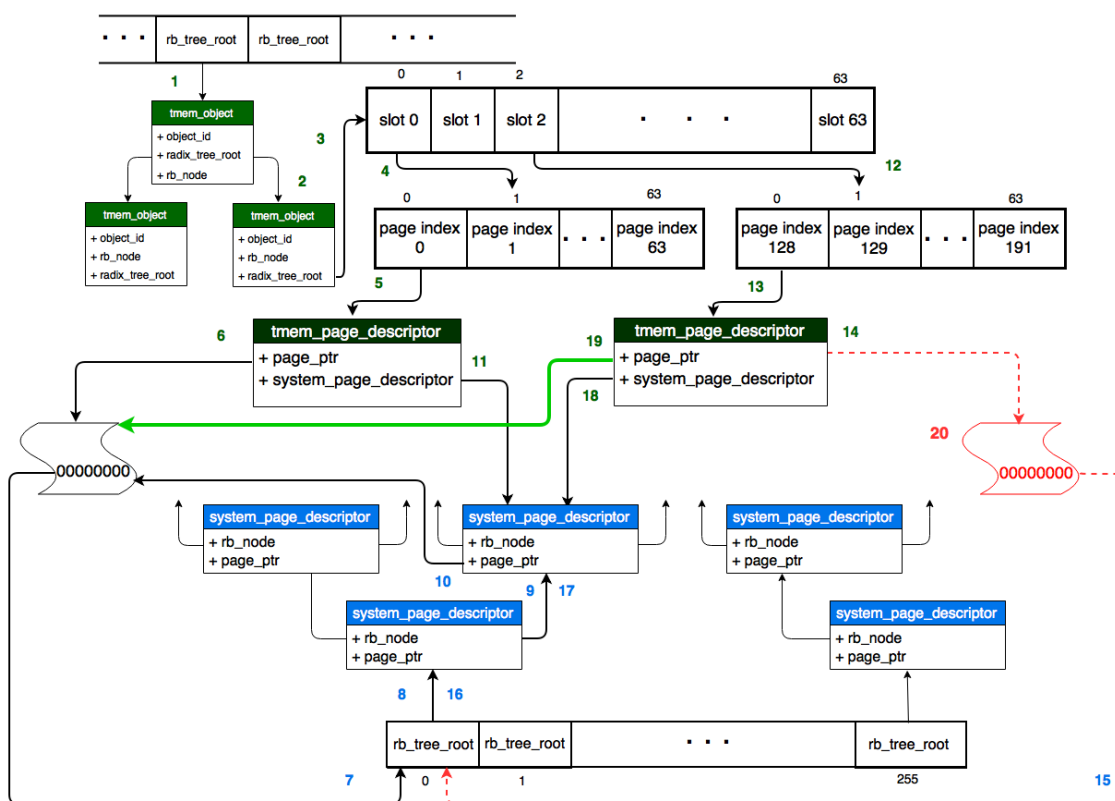


Figure 4.4: Tmem backend — VM & System view combined

#### 4.3.4 Local de-duplication

Xen makes use of synchronous or in-band approach to do local de-duplication. On a `TMEM_PUT_PAGE(pool_id, object_id, page_id, pfn)` call, the put operation is not immediately completed and returned. After copying the contents of the guest page with index `page_id`, of object `object_id`, to a newly allocated tmem page in the VM's `pool_id` pool and making the page pointer within the tmem page descriptor (4.3.3) to point to this newly created page, the Xen implementation traverses the `rbtree` in the system/hypervisor view. This traversal is done to check if a similar page is already present in the system or not. If such a page does exist then the page pointer of tmem page descriptor is updated to point to this already existing page and the recently created page is freed. On the other hand if no page with similar content could be found then a new system page descriptor (4.3.3) is created and its page pointer is made to point to the newly created page. The the algorithm for local de-duplication is given as algorithm 1. Refer to figure 4.4 for an illustration.

#### 4.3.5 Recap: local de-dup

Before the exploring the remote de-duplication design and implementation a recap of the tmem backend design and implementation that achieves local de-duplication.

- Objects in each VM's private pool can be looked up using a hash table, having 256 slots, indexed by a fixed length hash of the object id (the `inode number`) pointer. The object id is obtained from the handle in the tmem operation.
- Collisions in the hash table are resolved using `rbtree` that uses the fixed length hash of object id address, as the key, to structure itself.
- An `rbtree_node` is embedded in every object and that is how the object becomes a part of `rbtree`.
- Embedded within each object is a `radix tree` of pages belonging to that object. The key to `radix tree` is the index from the handle. The index is essentially the page number within that object.
- The leaves of the radix tree, in turn, point to a descriptor that describes a page in tmem pool. It has, among other things, a pointer to the actual system (host or hypervisor) memory page (RAM page frame) that holds the client page's data.
- The tmem page descriptor also points to another important structure called the

system page descriptor. The system page descriptor is the data structure used for enabling the sharing of pages, with similar content, among different client VMs.

- The system page descriptor are unique throughout an entire physical system. And it also points to the same system memory page, which was pointed to by the tmem page descriptor previously, that actually holds the client page's data.
- While the tmem page descriptor sums up a particular VM's view of the page in its private tmem pool, the system page descriptor is tmem backend's view of all the pages that have been put into the backend store by all client VMs. Both point to the same system memory page frame.
- The system page descriptor can be easily looked up by using a hash table indexed by the first byte of the contents of the client page it is holding. There is going to be only one system page descriptor hash table for the entire tmem backend.
- Collisions in this hash table are resolved using another rbtree which uses, as its key, the entire content of the page frame pointed to by the system page descriptor.
- The above description of the tmem backend is sufficient to understand how a look up happens on a client get request.



---

**Algorithm 1:** *put* in a de-duplicated tmem backend

---

```

1 if object already exists then
2   | (a) create a leaf node in the radix tree corresponding to the index of the
   | page and make it point to a newly created tmem page descriptor;
3 else
4   | create a node in the rbtree to which the object belongs and do (a);
5 end
6 copy the contents of the client page into the system page pointed to by the tmem
  page descriptor;
7 now with the first byte of the page content available with tmem find out the
  rbtree root in the system view in which the system page descriptor
  corresponding to this page can be found if present;
8 look for a match by comparing the data pointed to by the tmem page
  descriptor and the data pointed to by each system page descriptor;
9 if a match is found then
10  | free the system page pointed by the tmem page descriptor and make
   | the tmem page descriptor point to the system page pointed by the
   | matching system page descriptor;
11  | increment a reference count for the system page descriptor whose
   | content matched;
12 else
13  | insert a new system page descriptor into the system's view rbtree
   | and make the system page descriptor point to the system page pointed
   | by the tmem page descriptor;
14 end

```

---

**Note:** While doing a tmem flush/invalidation operation, if the `system page descriptor reference count` is not zero then neither the associated page is freed nor the `system page descriptor` is removed. Only the view of the client issuing a flush/invalidation operation is changing i.e. only the `tmem page descriptor` corresponding to the client is removed. (Xen, 2015)

# Chapter 5

## Design — co-operative de-duplication

### 5.1 Challenges of doing co-operative de-duplication

Before the design for co-operative de-duplication can be discussed the challenges posed by such a design should be discussed. A tmem backend that is going to look up other backends (remote backends), across the network, rather than looking up it's own local disk, in response to a `get` obviously works under the assumption that network access of memory is faster than than network disk access. This is an assumption that should be verified. Following are the list of major challenges involved with the design:

- (a) When there are more than one co-operating backends involved, issuing multicast requests to every other backend for a `put` and a `get` is going to increase the network traffic and add to the overheads of latency. Fan *et al.* (2000) have shown that the network communication and CPU overheads increase quadratically when total number of co-operating hosts go up. If for each host the average cache hit ratio is  $H$ , no. of hosts is  $N$ , average number of requests received is  $R$  then each host will have to handle  $(N - 1)(1 - H)R$  requests, and in the mesh of backends there will be  $(N)(N - 1)(1 - H)R$  requests in all.
- (b) The question of when to attempt a remote de-duplication is very important. Attempting to de-duplicate synchronously, in response to each `put` request, can cause inordinate delays to the client issuing the `put` request.
- (c) Ensuring consistency and maintaining coherency of data among the different views and hosts.

## 5.2 Co-operative or Remote de-duplication — design & implementation

This thesis draws heavily from the ideas presented by Fan *et al.* (2000) to solve the challenges that were posed in section 5.1. It should be recalled from (4.3) that all de-duplication activities happen in the *system/hypervisor view* and at that level the tmem backend implementation deals with a structure called `system page descriptor` (4.3.2).

### 5.2.1 Reducing network traffic using summaries

This thesis adopts the idea of storing summaries of the the contents of every other remote co-operating tmem cache to reduce the network traffic caused by multicast queries and responses when trying to do remote de-duplication. The way in which these summaries are put to use is different as this thesis solves a different problem than Fan *et al.* (2000). The use of summaries here is tightly coupled with remote de-duplication as mentioned. To remotely de-duplicate a page the local backend should know which remote backend possibly has this page. The obvious way to check if a page in the local tmem backend is available in any of the co-operating remote tmem backends is to multicast a query to all of them. This approach can give rise to about  $(N)(N - 1)(1 - H)R$  requests in the network of backends (refer 5.1).

#### 5.2.1.1 Remotification — remote de-duplication of a local page

To avoid problem of proliferation of remote de-duplication requests in the network summaries of remote backend contents are used. First consult these locally available summaries. If a particular summary suggests a possible presence then send the actual page across to that backend, do a byte by byte comparison there, confirm that it is an actual match. After the confirmation update the *system view* (refer 4.3.2) of tmem by adding a reference to the remote page in the `system page descriptor` corresponding to this page and free the local system page pointed by the `system page descriptor`. At this point the page is marked as *remotified*. This way remote de-duplications are hidden away from guests' view of tmem.

#### 5.2.1.2 Challenges posed by the use of summaries

The use of summaries itself pose another set of challenges.

- (a) If the number of co-operating remote hosts goes up then there can cost-benefit trade off between the amount of memory used to store the remote summaries and the reduction in network traffic achieved.
- (b) The frequency of summary updates required to avoid staleness of the information held in summaries is another issue. Since the summaries cannot give a real time snapshot of the remote contents or the changes that are happening to the remote contents the tmem implementation should keep refreshing it. The frequency of refreshing the remote summary depends on how tolerant is the tmem implementation i.e indirectly how tolerant are the frontends to
  - *False misses* — which occur when a page is available at a remote backend but that information could not be reflected in its summary held locally. This is lesser of the two *evils* as this translates only to a missed opportunity to remotify a page or to de-duplicate a page remotely. A *false miss* does not result in any network traffic.
  - *False hits* — which occur when the summary wrongly indicates the presence of a page in a remote backend when actually it is not present. This results in unnecessary network traffic in the form of wasted remote de-duplication requests.

due to staleness in the information held summaries. But none of these affect the correct working of the frontends (or backends) because remote de-duplication attempts are only an optimization and not a requirement for correct working.

### 5.2.1.3 Using bloom filters to represent summaries

In order to address the challenges of using a summary this thesis takes the same route as of Fan *et al.* (2000). They made use of a data structure called bloom filters to represent remote cache summary. Bloom filters are probabilistic hash based data structures that can represent a set of keys with minimal memory requirements and answer membership queries in it with zero probability for "false negatives" and low probabilities for "false positives" i.e. a bloom filter will say respond with a "possibly in the set" or "definitely not in the set" to a membership query. The basic idea behind the working of bloom filters is explained below (Wikipedia, 2016).

1. allocate a vector  $v$  of  $m$  bits. Initially all bits of this vectors are set to 0.

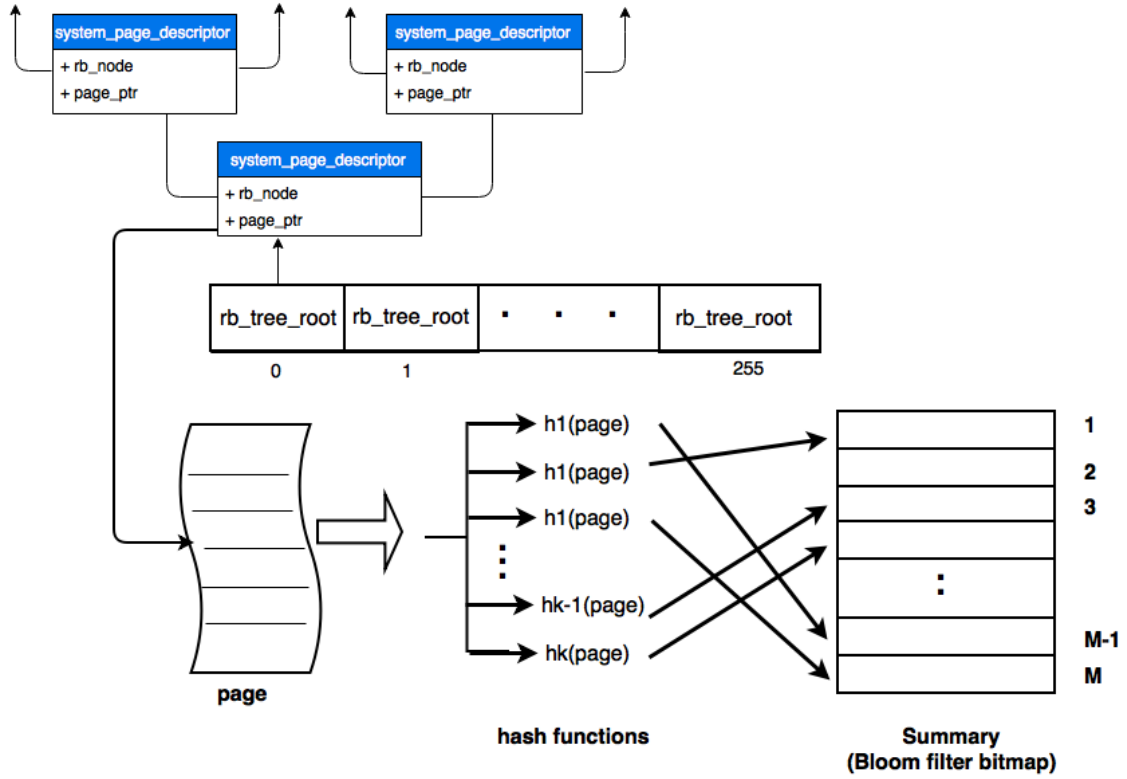


Figure 5.1: Setting bloom filter bits on a tmem put

2. select  $k$  independent hash function  $h_1, h_2, h_3, \dots, h_k$  each having a range  $\{1, \dots, m\}$ .
3.  $\forall$  elements  $a \in \text{set } A$  of our interest, the bit positions corresponding to the  $k$  hash functions  $h_1(a), h_2(a), \dots, h_k(a)$  in the bit vector  $v$  are set to 1. This marks the presence of that element  $a$  in the set  $A$ . It should be evident by now that multiple elements belonging to the set  $A$  can turn on the same bits in the vector  $v$
4. To search for an element  $b$  in the set  $A$  check if the bit positions  $h_1(b), h_2(b), \dots, h_k(b)$  are set to 1. If all are 1s then there is a certain probability that element  $b \in A$ . But as other elements in  $A$  can also set the same bits as  $b$  there is a certain probability of getting a “false positive”. On the other hand even if any of  $h_1(b), h_2(b), \dots, h_k(b)$  is not set then it can be safely concluded that  $b$  is not present in  $A$ .

To represent tmem backend summaries using bloom filters first we need to specify a bitmap  $v$  and select  $k$  hash algorithms. Now when a unique page is put into the tmem backend for the first time by a frontend after step 13 of textttalgorithm 1 the *system view* will access the page using the reference available in `system_page_descriptor` and the  $k$  hash functions are applied on the contents of the page. The corresponding  $k$  bit positions

in the bitmap  $v$  are set to 1. Thus the bloom filter is populated by repeating these steps each time an unique page is put into the tmem backend for the first time. And before a page is removed from the backend the  $k$  bits corresponding to this page are unset. The bloom filters then have to be sent to every other co-operating backends. Refer figure for illustration 5.1

Once the bloom filters of all the co-operating backends are received a backend can start querying them locally as mentioned in 5.2.1.1 to explore remote de-duplication opportunities.

#### 5.2.1.4 Issues with using bloom filters as tmem cache summary

If the  $k$  bits corresponding to a page are unset from the bitmap  $v$  when the page is removed from the backend the bits set by other elements are also removed. This can cause “false negatives”. Fan *et al.* (2000) suggests associating a counter with each bit position in bitmap  $v$  to ensure that a bit position is not unset unless the counter value goes to zero. This counter should be of a data type having large size to count the large number of pages that can set a bit position. If  $m$ , the total number of bits in  $v$ , is a big number then using a counter for each bit in  $v$  will lead to an increase in the memory consumed bloom filter. Hence this thesis chose not to unset the  $k$  bit positions in  $v$  when a page is removed from the backend, thereby avoiding “false negatives” completely and preferred to suffer from more “false positives”.

The probability of “false positives” are related to  $m$ . The larger the bitmap the smaller the number of false positives and the inverse is also true. The “false positive” probability is given by  $(1 - e^{-kn/m})^k$ . Hence a lot of adjustments can be done by varying the values for  $n, k, & m$  to get a tolerable “false positive” probability. The following table depicts the various “false positive” probabilities for different values of these parameters (Mill, 2010). The “false positive” probability possible for the experimental settings used in this thesis, assuming each VM in the hypervisor is going to dump its entire memory contents into the tmem backend, is shown in figures 5.2 & 5.3.

Memory of a VM (GB)	Pages	Summary pages (N)	Bit slots in bloom filter (M)	M (MB)	Total size of all bloom filters (GB)	Hash functions (K)	Probability of false positive $(1-e^{(-KN/M)})^K$
0.5	131,072	2,097,152	268,435,456	32	0.5	2	0.0002
						4	0.0000008
						8	0.0000000001
			33,554,432	4	0.0625	2	0.0138
						4	0.0024
						8	0.0006

Figure 5.2: Bloom filter stats assuming each guest is going to dump its entire max configured memory contents of size 0.5 GB into tmem cache. VMs per host are 16, total co-operating hosts are 16

Memory of a VM (GB)	Pages	Summary pages (N)	Bit slots in bloom filter (M)	M (MB)	Total size of all bloom filters (GB)	Hash functions (K)	Probability of false positive $(1-e^{(-KN/M)})^K$
4	1,048,576	16,777,216	268,435,456	32	0.5	2	0.0138
						4	0.0024
						8	0.0006
			33,554,432	4	0.0625	2	0.3996
						4	0.5590
						8	0.8625

Figure 5.3: Bloom filter stats assuming each guest is going to dump its entire max configured memory contents of size 0.4 GB into tmem cache. VMs per host are 16, total co-operating hosts are 16

The frequency of exchange is another important decision that is to be made to avoid staleness of the information present in bloom filters as pointed out in 5.2.1.2. For a page in tmem backend if a “false miss” occurs due to the staleness in all the locally available summaries only an opportunity to do remote de-duplication of that page is lost. But, for a page, if a “false hit” occurs in any one of the remote summaries available then a de-duplication request is to be sent across to the owner of this summary. This request contains a page to do actual byte-by-byte comparison at the remote host. All this effort is wasted as the remote backend is going to respond in negative as it was a “false hit”. Thus not only a remote de-duplication attempt failed but also added to the network overhead. In the experimental and correctness evaluations conducted in this thesis the frequency of bloom filter exchanges were varied to suit the tests being carried out at that instance.

### 5.2.2 Network setup for bloom filter forwarding & remote de-duplication

A choice made to avoid the latency that can arise because of exchanging large bloom filters was to have a dedicated *leader server* that does the job of distributing the bloom filter of one server (tmem backend) to all other servers that are currently online. When a server comes up it registers with the *leader server* who maintains a list currently active online servers 5.4. A kernel thread within all servers other than *leader server* was assigned the task of forwarding bloom filter periodically to the leader server and the periodicity of the transfer, which is a configurable parameter, is set to 2 mins as a general setting. The *leader server* then serially unicasts it to all the other online servers thus taking this burden off a server. Within the *leader server* each online server is treated as a separate thread.

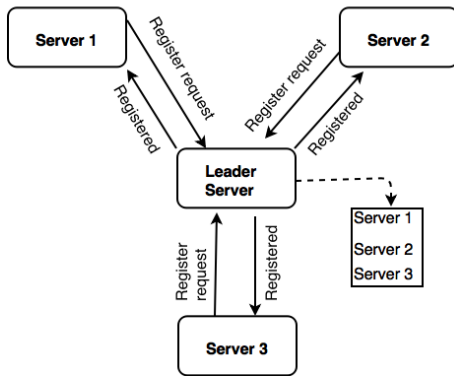


Figure 5.4: Tmem backends registering with *leader server*

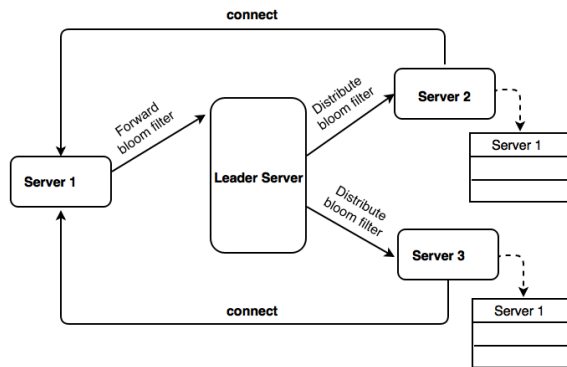


Figure 5.5: bloom filter exchange

A server (backend) will take cognisance of another server only when the former receives the latter's bloom filter. On receipt of a bloom filter the former will open up a connection with the latter and this connection will last till former exits 5.5. This connection is to do all other communications specific to doing remote de-duplication like — (a) remote de-duplication attempts (remotification requests) (b) retrieving the contents of a remotely de-duplicated page (remotified get) — directly with the latter without involving the *leader server*. Refer 5.10 & 5.11. A server has one thread each for every other server to handle such queries from others. A server's exit message is passed onto the *leader server* to be transmitted to others. A server on coming to know



from the *leader server* of another server's exit will close the persisting connection it had with it. All network communications use `tcp` and the implementation handles possible cases of graceful exits of *leader server* and/or others'. Distributed concepts like *keep-alive heartbeats* etc were not implemented in order to keep the implementation simple and due to shortage of time. **Note:** These choices of limited features were made as the objective of this thesis have more to do with evaluating the overheads of doing remote de-duplication rather than getting the best settings for enabling-functionalities like bloom filters, network communication etc.

### 5.2.3 Deciding when to do *remotification*

This section tries to answer the question of deciding when to attempt a remote de-duplication 5.1. This is easy to comprehend from the narrative so far that it is not a wise choice to attempt a de-duplication synchronously along with a `put` from VM. Another fact affecting the decision of when to attempt a remote de-duplication is the availability of memory within the hypervisor. It doesn't seem logical to *remotify* i.e. remote de-duplicate a page if there is enough memory available in the hypervisor. If the hypervisor is experiencing memory pressure it can evict pages from the `tmem` backend. To have a control over how much and what to evict the `tmem` backend implementation can register a shrinker with the hypervisor like `zcache`. Taking into account these considerations this thesis implemented a simple eviction thread in hypervisor (kernel) that evicts pages from the `tmem` backend — (a) if the total number of pages in backend crosses a statically configured threshold **or** (b) whenever dynamic eviction is enabled by writing to a `debugfs` entry. The number of pages to be evicted when dynamic eviction is enabled should be specified as another `debugfs` entry. Hence *remotification* of `tmem` pages is coupled with memory pressure and eviction.

### 5.2.4 Additions to local de-duplication to enable remote de-duplication

The additions to be done to the local de-duplication algorithm in the `put` path to enable co-operative de-duplication are listed in the following algorithm. The steps of importance are steps 4 and 8 of algorithm 2. Refer figure 5.6 for an illustration.

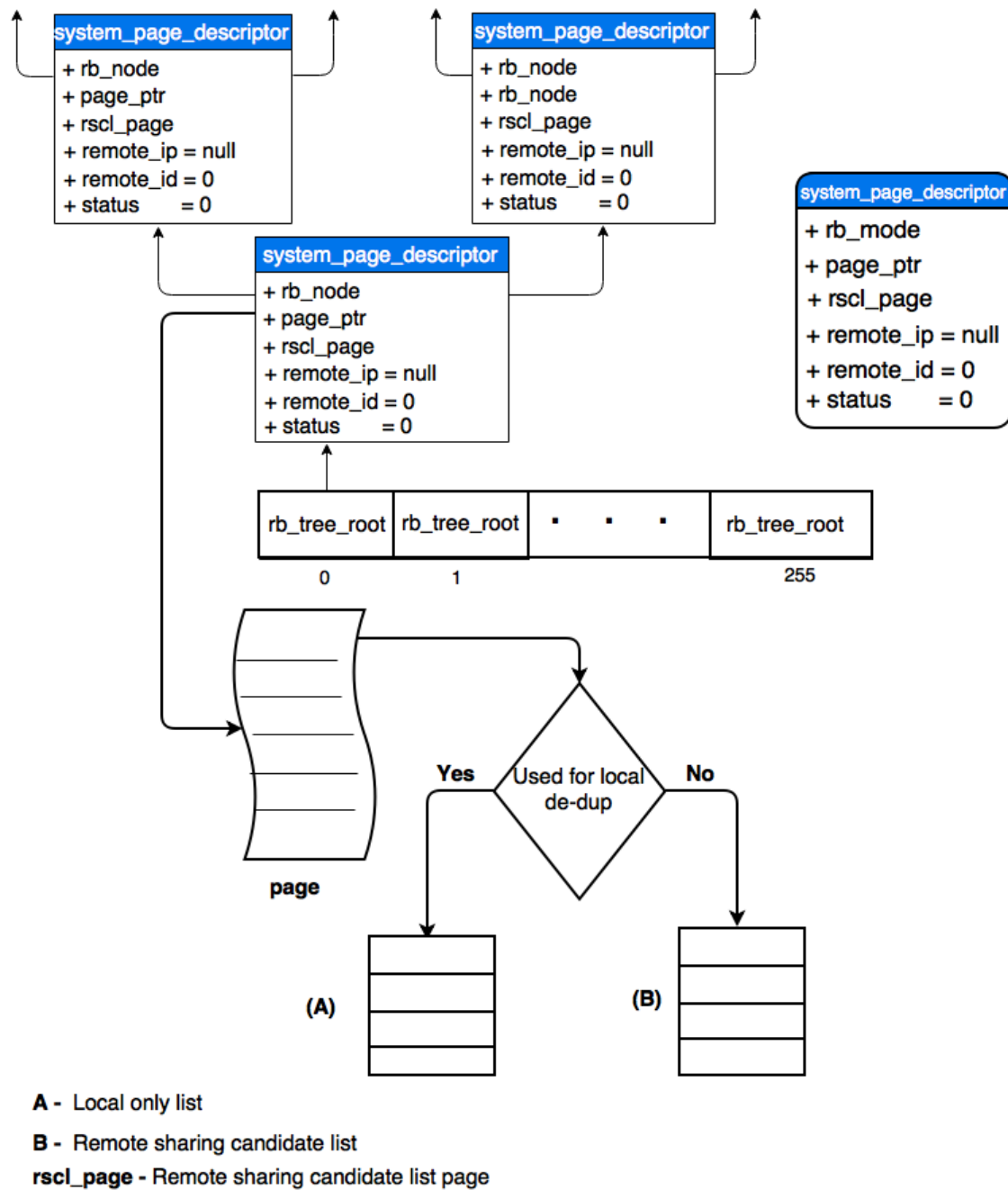


Figure 5.6: extending local de-dup to use *remote sharing candidate list* & *local only list* to support remote de-dup

---

**Algorithm 2:** additions to a *put* in a tmem backend to enable co-operative de-duplication

---

```

1 if local sharing possible then
2     hold on to the page in the local tmem backend by making the tmem page
      descriptor page pointer point to the already existing local page pointed
      by system page descriptor;
3     // steps 6, 7, 8, 10, 11 of Algorithm 1
4     add this existing system page descriptor to a local only list;
5 else
6     explicitly store the page locally in tmem;
7     // steps steps 6, 7, 8, 13 of Algorithm 1
8     add the new system page descriptor a remote sharing candidate list;
9 end
10 as per the remote sharing policy devised, look for remote de-duplication;
```

---

### 5.2.5 Eviction & remotification

Section 5.2.1.1 had provided the basic steps involved in carrying out *remotification* of page. This section explains the same with specifics from the tmem backend implementation. One thing to be noticed here is that eviction is attempted only from the *remote sharing candidate list* as preference is given to a page that is being shared and would not be evicted from the backend. Even after finding a remote match the page is not removed from the backend and the corresponding system page descriptor moved to a *remote shared list* immediately. Instead the backend makes sure that this page was not used for local de-duplication of another page or an attempt to delete this page was not done while the eviction thread was searching for a remote match. If it was used for the de-duplication of another page then the system page descriptor is moved from *remote sharing candidate list* to *local only list*. And if a deletion was attempted on this page, then the system page descriptor is deleted and the system page is freed. The following algorithm explains what happens when eviction thread is fired.

**Algorithm 3: eviction** in a tmem backend that does remote de-duplication

---

```

1  foreach system page descriptor in the remote sharing candidate list do
2      look up the locally stored remote backend summaries for a possible match;
3      foreach remote backend summary do
4          if match found in a remote backend summary then
5              confirm the match by querying that particular remote backend and
              doing an entire page comparison on the remote host;
6              if local page contents matches with remote page contents then
7                  ensure that in the meanwhile neither a local de-duplication
                  using this system page descriptor nor a deletion of this
                  system page descriptor was attempted.;
8                  if local page was used for a local de-dup then
9                      declare the remotification attempt failed;
10                     move the system page descriptor from remote sharing
                     candidate list into a local only list;
11                     break;
12                 end
13                 if an attempt to remove the local page was done then
14                     declare the remotification attempt failed;
15                     remove the system page descriptor from remote
                     sharing candidate list;
16                     delete the system page descriptor and the system page
                     from the backend;
17                     break;
18                 end
19                 move the local system page descriptor from remote
                     sharing candidate list into a remote shared list;
20                 update the system page descriptor fields to make it point
                     to the system page descriptor in remote backend.;
21                 free the local system memory page frame;
22                 break;
23             else
24                 repeat step 7;
25                 // Repeat applicable steps from 9,10,14,15,16
26             end
27         else
28             continue;
29         end
30     end
31 end

```

---

**Note:** Step 20 of algorithm 3 instructs to update the local `system page descriptor` fields to make it point to the remote `system page descriptor` in a remote backend as a result of finding a remote page (pointed by the remote `system page descriptor`) to de-duplicate the local page (pointed by the local `system page descriptor`) with. This update includes changing `status` to `remotified`, changing `remote_ip` to the ip of remote backend where match was found, updating `remote_id` to value returned by remote backend, setting the page pointer to NULL.

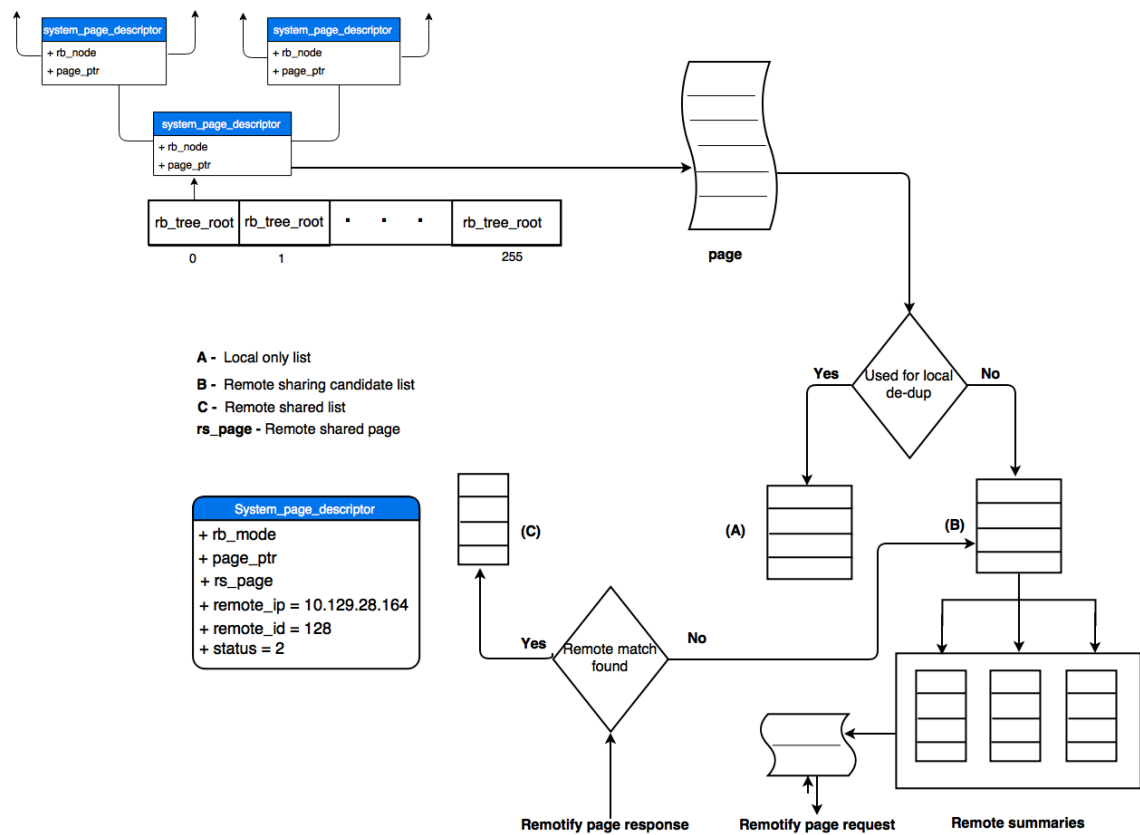


Figure 5.7: use of *remote shared list* to keep references to pages that were successfully remote de-duplicated

### 5.2.6 Responding to a *remotification* attempt

This section explains the actions taken by a remote backend when it receives *remotification* request from another backend. The approach is simple. In the backend's *system view* find the `rbtree` based on the first byte of page that came with the request. Within this `rbtree` look for a `system page descriptor` that could be pointing to a page with similar content. This is done by doing a byte-by-byte comparison

of request page with the page pointed to each system page descriptor. On finding a match a local page is added to a radix tree called *remote radix tree* and its index within that radix tree is returned along with the response to the *remotification* request. This ensures that a subsequent *remote get* for this page, from the backend which just did the *remotification*, can be served without having to search through all *rbtrees* in the *system view*. In fact there is not just a single such *remote radix tree* but 256 different ones, with their own private indices starting from 0, whose roots are accessed via the first byte of a page. This is the same approach adopted in 4.3.1.1 to prevent a lone tree from growing too large. The fact that a page is being used for a remote de-duplication does not prevent it from being deleted from a backend if the local reference count of the containing system page descriptor goes to 0 as a result of all local VMs, that accessed it earlier, withdrawing their references to it.

---

**Algorithm 4:** *remote put* in a tmem backend as a response to *remotification* attempt

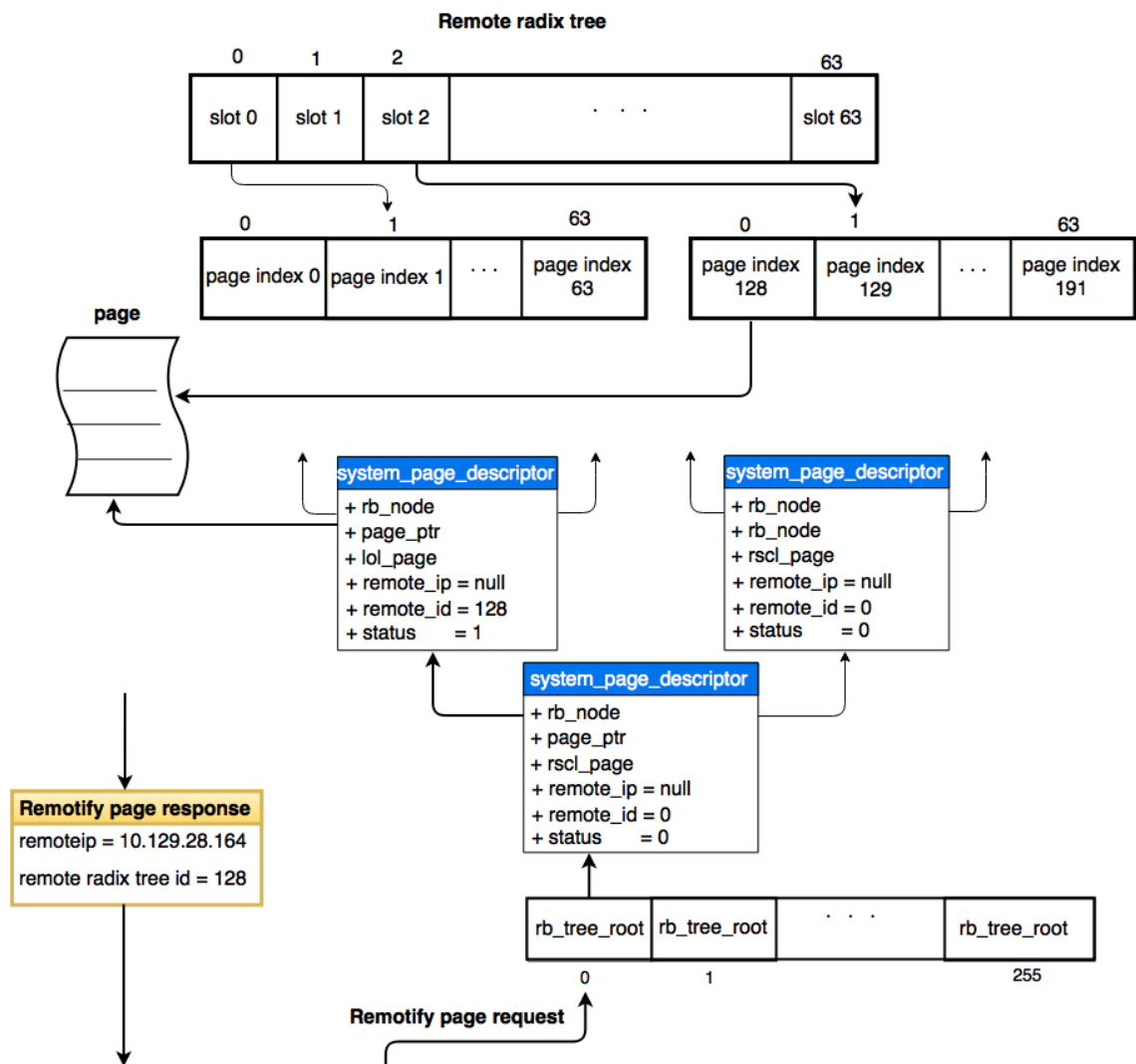
---

```

1 foreach system page descriptor in the system view of tmem backend do
2   compare the contents of the local page pointed by this system page
   descriptor with the contents of the page that came with remotification
   request;
3   if matching then
4     add the local page pointed by the this system page descriptor to a
     remote radix tree. The root of this remote radix tree can be
     accessed by the first byte of page content;
5     return index of this page within this remote radix tree and the
     backend's ip address;
6     // this references returned are used by step 20 of
       Algorithm 2 as pointed out in note 31
7   else
8     return null;
9     // takes you to by step 28 of Algorithm 2
10  end
11 end

```

---

Figure 5.8: Illustration of remote de-dup in response to a *remotification* request

### 5.2.7 Remotified get — retrieving a remotified page

When VM issues a get for a page (that it had previously put) from tmem that was remotified without the VMs knowledge, an attempt to get the contents of this page from remote tmem backend is attempted. This is called a *remotified get*. The *system view* will come to know from the *system page descriptor* that the page pointed by it is *remotified* and using the credentials available with the *system page descriptor* it queries the remote backend. If the page was removed from the remote backend because no more local references to it, from any of the VMs there remained, then the backend that issued this *remotified get* will have to fetch the page from a network disk. This approach

suffers from too much latency as it adds the overhead of an extra network operation. Realizing this the algorithm of a *remote put* (algorithm 4) that happens as a response to *remotification* can be modified to add a step after step 4 that increments the refcount of a local system page descriptor that was used to de-duplicate another remote page. Once the page is found in the remote backend it is copied and given back as a response to *remotified get* attempt.

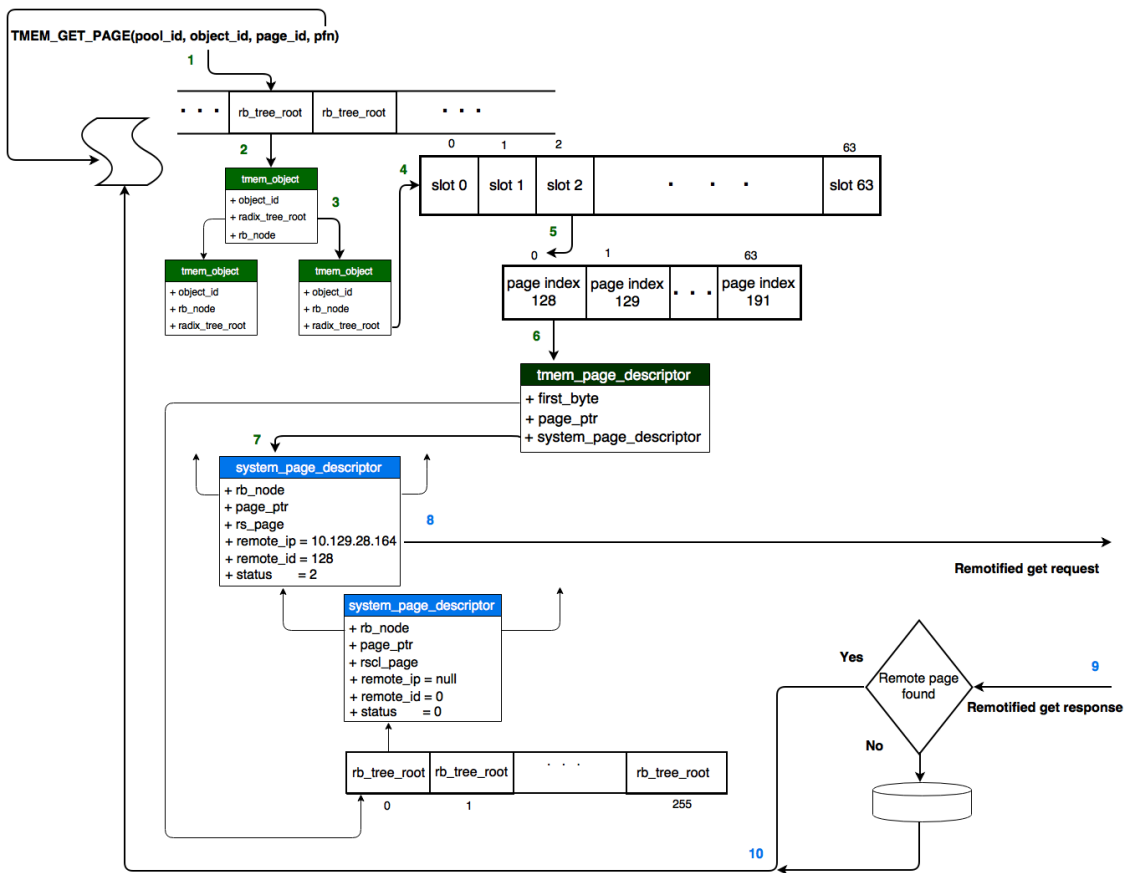


Figure 5.9: Illustration of a *remote get* request for a page from a remote backend in which this page was de-duplicated

### 5.3 Summary

Thus over the past two chapters the design and implementation aspects of a tmem backend that does local and co-operative de-duplication by making use of separate views of the tmem backend cache for the VMs and the hypervisor was explored.



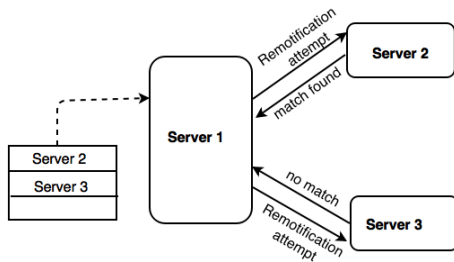


Figure 5.10: *remotification* - attempt remote de-dup

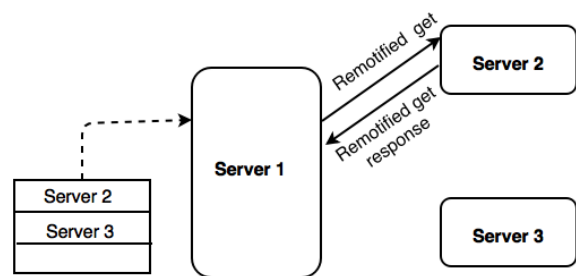


Figure 5.11: *remotified get* - retrieving contents of remote de-duplicated page

# Chapter 6

## Experiments & Evaluations

This chapter gives the details of the experiments and evaluations that were conducted to – (a) verify the correctness of implementation (b) evaluate the end-to-end delay experienced by a synthetic read intensive workload (c) micro-benchmark the execution times of the important functions in the tmem implementation in terms of number of CPU cycles spent on each. Experiments for (b) were conducted under 3 settings — (1) without the tmem cache enabled (2) with tmem cache enabled but doing only local de-duplication and (3) with the tmem cache enabled and doing both local as well as remote de-duplication.

### 6.1 Experimental Setup

The experimental setup consisted of 3 machines. Machine A with 4 Intel Core i5-4440 3.10 GHz CPUs, 8 GB RAM; machine B with 4 Intel Core i7-3770 3.40 GHz CPUs, 8 GB RAM; machine C with 4 Intel Core 2 Quad Q9550 2.83 GHz CPUs, 6 GB RAM. All three booted a custom Linux kernel based on linux-4.1.3. Machines B and C hosted 1 VM each using KVM. The VMs were configured to boot with 512 MB RAM and had all other configurations same.

### 6.2 Correctness Verification

This section presents the correctness tests that were carried out. The correctness verification tests are divided into verification of — (a) Local de-duplication (b) Remote de-duplication.

### 6.2.1 Local De-duplication

#### 6.2.1.1 Question

Whether duplicate pages are being correctly identified and the redundant copies being correctly removed from the tmem backend store.

#### 6.2.1.2 Experiment design

Only Machine A with a single VM running in it was taken for this experiment. A file in the VM consisting of all 1s were created in the VM — the *one file*. The size of *one file* was 1 GB, while the maximum memory allocated to this VM was 512 MB. The main workload of this test is synthetic, one of copying this 1 GB *one file* to `/dev/null`. The steps of the tests are as follows:

1. Insert the tmem backend module, boot the VM.
2. Obtain the debugging information from `syslog` at the host.
3. Run the workload in VM.
4. From the collected debugging information find out the number of pages of the *one file* put into the tmem backend and the number of pages that were successfully de-duplicated. There is provision for using the `inode` of the *one file* in VM for this purpose.

#### 6.2.1.3 Observations & conclusions

It was observed that the number of pages of *one file* that successfully de-duplicated was exactly one less than the number of pages that were put into the tmem backend. Since the configured memory size of the VM (512 MB) is much less than *one file* size (1 GB) when the workload is run there is going to be memory pressure inside the VM and this will trigger evictions of clean pages from the page cache. These pages will be put into the clean cache. And as only one page of the *one file* was retained in the tmem cache and rest all pages were de-duplicated using this file it can be concluded that local de-duplication works as expected.

## 6.2.2 Remote de-duplication

### 6.2.2.1 Question

Whether matches for pages coming in from a remote backend can be found out and de-duplicated. Also check whether all such pages that were *remotified* be retrieved successfully. i.e. check the correct working of *remotification* and *remotified get*.

### 6.2.2.2 Experiment design

Machines B & C were used in this verification. Both host a single VM that are identical. A file of size 1 GB with random content was created in the VM running in one of the machines and copied to the VM in the other — the *rand file*. The workload of this test is similar to the previous experiment. A synthetic workload where this *rand file* is copied to */dev/null*. Evictions are disabled in the tmem backend of Machine B so that it retains all the pages that were put into it. The steps of the tests are as follows:

1. Insert the tmem backend module and boot the VMs in both the machines.
2. Run the workload in both. Since the configured memory sizes of the VMs (512 MB) are much less than *rand file* size (1 GB) when the workload is run there is going to be memory pressure inside the VMs and this will trigger evictions of clean pages from the page caches to tmem caches.
3. Flush the pages caches of both the VMs. This will ensure that any residual pages of the *rand file* in the VM page caches will be evicted to the tmem caches.
4. The bloom filters of both machines B and C would have marked the presence of all these pages.
5. Fire the bloom filter exchange threads in both machines.
6. Fire the eviction thread in machine C. This will lead to all pages of the machine being evicted, explored for remote de-duplication in machine B.
7. Observe the total evictions (*remotification*) happening from machine C for the pages of the inode corresponding to *rand file*. Also observe the number of remote de-duplications happening for pages of this same inode in machine B. There are provisions to do these in the form of debugfs entries in each machine.
8. Run the workload in machine C again.
9. Observe the total *remotified get* requests originating from machine C for the pages of the inode corresponding to *rand file*. Also observe the number of *remotified get*

requests served for pages of this same `inode` in machine B. There are provisions to do these in the form of `debugfs` entries in each machine.

### 6.2.2.3 Observations & conclusions

Since machine A is not evicting any of the pages it is holding in the `tmem` cache all pages corresponding to the *rand file* that got put in it from the VM during the workload run and during page cache flush at the VM will remain in machine A's `tmem` cache. Once this information is exchanged among both the machines via `bloom filters` machine C will find matches, for all the pages of *rand file* that it is evicting from its `tmem` cache, in machine B. Also machine C should be able to retrieve all the pages that it *remotified* in this manner. It was observed in:

- step 7 of procedure 6.2.2.2 that the total evictions (*remotification*) happening from machine C for the pages of the `inode` corresponding to *rand file* was exactly equal to the number of remote de-duplications happening for pages of this same `inode` in machine B. Thus verifying the correctness of remote de-duplications.
- And in step 9 it was observed the total *remotified get* requests originating from machine C for the pages of the `inode` corresponding to *rand file* was exactly equal to the number of *remotified get* requests served for pages of this same `inode` in machine B. Thus verifying the correctness of *remotified gets*.
- Also both these numbers were equal to the entire file size of *rand file*.

### 6.2.2.4 Related questions

A couple of similar test as to 6.2.2.1 were also carried out. The first of these were to verify that if none of the pages of *rand file* were *remotified* from `tmem` cache of machine C then none would be retrieved from machine B even if those were available in the machine B's `tmem` cache. It was found that none of the pages of *rand file* were retrieved from machine B.

The second experiment verified whether a page can be wrongly de-duplicated in a remote backend even if such a similar page did not exist there. When machine B was not holding any pages of *rand file* machine C tried to *remotify* the pages from its copy of *rand files* available in its `tmem` cache. The results showed that machine C was unsuccessful in *remotifying* even a single page.

These experiments illustrates the correct working of the eviction thread as no page that was not *remotified* by by a backend will be retrieved from a remote backend and no page that is not available in a remote backend will be remote de-duplicated in that remote backend.

#### 6.2.2.5 Question

The test 6.2.2.1 was done with machine B opting not to remotify. The following test was done to verify whether the implementation will work correctly when both machines are working in same mode with evictions enabled.

#### 6.2.2.6 Experiment design

Two 1 GB random files were created in one of the VMs — X & Y. They were copied to the other. This synthetic workload is about both VMs copying X & Y to /dev/null. Following this both VMs flush their page caches. And they exchange their bloom filters. Machine B evicts pages of file X from its tmem cache, while machine C evicts pages of file Y from its tmem cache. This is followed by VM in machine B repeating the workload only for file X and VM in machine C repeating for only file Y.

#### 6.2.2.7 Observations & conclusions

1. It was found that that the total evictions (*remotification*) happening from machine B for the pages of the `inode` corresponding to file X was exactly equal to the number of remote de-duplications happening for pages of this same `inode` in machine C. Thus verifying the correctness of remote de-duplications for file X *remotified* from machine B.
2. And it was observed the total *remotified get* requests originating from machine B for the pages of the `inode` corresponding to file X was exactly equal to the number of *remotified get* requests served for pages of this same `inode` in machine C. Thus verifying the correctness of *remotified gets* for file X *remotified* from machine B
3. The symmetrical cases obtained by substituting X with Y and B with C too were found to be holding.

### 6.3 Evaluation of runtime overheads

Recall that when a VM access a file, its own page is searched for the required pages. This is provided the access did not happen with a `O_DIRECT` flag. If the pages of the file block were not found then a block layer disk IO operation is occurs. If the pages were hit in the page cache the above mentioned disk IO can be avoided. In addition when `tmem` is enabled when a page cache miss occurs a `tmem get` is issued to try and get the pages required. If it is a hit in the `tmem` cache the page is returned to the VM application. Else if the page was found to be *remotified* in the `tmem` cache then it has to be fetched from the remote `tmem` cache synchronously and served to the application in VM. If neither of this succeeds a disk IO is unavoidable.

This section explores the end-to-end runtimes experienced by a synthetic workload (the one used in section 6.2.2) in a VM using the `tmem` cache. The workload copies a 1 GB file of random content to `/dev/null`. This file, called the *rand file*, was made from the contents of `/dev/urandom`. This section compares the runtimes experienced by this read intensive workload when the `tmem` cache is hot against when it is cold. The experiment is run with 2 setting for `tmem` cache — one with only local de-duplication enabled, and the other with both local and remote de-duplication enabled.

#### 6.3.1 Runtime overhead with local only de-duplication

	<b>Cold</b>	<b>Hot</b>
Mean Runtime	15.305	2.464
Std Deviation	0.475	0.028

Table 6.1: Effects of hot & cold `tmem` caches in local de-dup only mode on a synthetic workload in a VM hosted in machine B. Runtime is specified in seconds. Base case mean 14.456s. The results were obtained over 100 runs.

##### 6.3.1.1 Inference

The mean runtime of the same workload on machine B without `tmem` cache being loaded is 14.456s. This also is the base case. When the `tmem` cache is cold a `get` request incurs the overhead of querying the `tmem` backend and still ends up going to the disk to get the actual page. This is the reason why the mean runtime of cold cache is greater than the mean runtime without caching at all. But when the workload is repeated after flushing

the page cache pages of the VM the runtime improves drastically, almost 6 times faster when compared to base case and slightly more 6 times faster than compared to cache cold scenario.

### 6.3.2 Runtime overhead with local & remote de-duplication

	<b>Cold</b>	<b>Hot</b>
Mean Runtime	15.724	32.759
Std Deviation	0.688	2.194

Table 6.2: Effects of hot & cold tmem caches in local & remote de-dup mode on a synthetic workload in a VM hosted in machine B. Runtime is specified in seconds. Base case mean 14.456s. The results were obtained over 100 runs.

#### 6.3.2.1 Inference

The base mean runtime of the same workload on machine B without tmem cache being loaded is 14.456s. When the tmem cache is cold a `get` request incurs the overhead of querying the tmem backend and still ends up going to the disk to get the actual page. This is the reason why the mean runtime of cold cache is greater than the mean runtime without caching at all. When the workload is repeated after flushing the page cache pages of the VM and doing *remotification* of pages belonging to the file used for the experiment by evicting them forcefully from the tmem cache, it is seen that the runtime becomes more than double compared to both the base case and the cache cold scenario. Even though a successful retrieval of a *remotified* page is counted as a cache `hit` and considered along with the cache `hot` case, it actually involves a over the network memory access which is obviously slower than local disk access. Hence the cache `hot` case has more mean runtime.

## 6.4 Micro-benchmarking of important functions in the backend implementation

Towards benchmarking the run-time overheads of individual operations, the CPU cycles for significant tmem backend operations are obtained from the tmem implementation. This excludes the overheads of `VMExit` and `VMEEntry` caused by the hypercalls. The following table 6.3 shows the CPU cycles consumed by operations that are significant when



the tmem functions in local de-duplication mode alone. Table 6.5 shows the CPU cycles consumed by significant operations when tmem runs with local & remote de-duplication enabled.

### 6.4.1 Experiment design

All the tests carried out in this section have a similar setting as of the previously done correctness verification tests. The benchmarking of tmem operations done by a tmem backend doing only local de-duplication has the same setting as 6.3 and the one both remote & local de-duplication makes use of the same setting as of 6.2.2.2.

### 6.4.2 Micro-benchmarking tmem operations in local de-dup only mode

Operation	CPU Cycles	Total #
kvm_host_get_page ( <b>hits</b> )	16143 $\pm$ 10257	263031
kvm_host_get_page ( <b>miss</b> )	8745 $\pm$ 1951	264676
kvm_host_put_page	20859 $\pm$ 7278	262752
local de-duplication ( <b>succ</b> )	0	0
local de-duplication ( <b>fail</b> )	8138 $\pm$ 2236	262752

Table 6.3: Table depicting the overhead of tmem functions in terms of CPU cycles when the tmem cache does only local de-duplication. Test used the *rand file*. get miss overheads were separately calculated without populating the tmem cache.

#### 6.4.2.1 Inferences

The get **hits** consume more cycles than get **miss** because it involves explicit copying of a page worth of data from a tmem cache to guest page. The same holds for put operation. Since the file used in the test was a file generated from /dev/null full of random content all de-duplication attempts were a failure even though the tmem cache held the entire content after the at the end of the first run.

Operation	CPU Cycles	Total #
kvm_host_get_page ( <b>hits</b> )	7440 $\pm$ 6037	263166
kvm_host_get_page ( <b>miss</b> )	9014 $\pm$ 2026	278397
kvm_host_put_page	23655 $\pm$ 9137	262753
local de-duplication ( <b>succ</b> )	10048 $\pm$ 2825	261381
local de-duplication ( <b>fail</b> )	8325 $\pm$ 4098	1372

Table 6.4: Table depicting the overhead of tmem functions in terms of CPU cycles when the tmem cache does only local de-duplication. Test used the *one file*. `get` miss overheads were separately calculated without populating the tmem cache.

#### 6.4.2.2 Inferences

The above table (6.4) shows the number of CPU cycles consumed by local de-duplication operations. The sum of successful de-duplication attempts & failed de-duplication attempts are equal to total puts. Other inferences are the same as 6.4.2.1.

### 6.4.3 Micro-benchmarking tmem operations in local & remote de-dup mode

Operation	CPU Cycles	Total #
kvm_host_get_page ( <b>hit</b> )	379232 $\pm$ 387284	258503
kvm_host_get_page ( <b>miss</b> )	0	0
kvm_host_put_page	107007 $\pm$ 241533	258036
remotify_page ( <b>succ</b> )	960450 $\pm$ 76539	155316
remotify_page ( <b>fail</b> )	0	0
remotified_get_page ( <b>succ</b> )	623392 $\pm$ 305216	152256
remotified_get_page ( <b>fail</b> )	0	0

Table 6.5: Table depicting the overhead of tmem functions in terms of CPU cycles when the tmem cache does both local & remote de-duplication.

#### 6.4.3.1 Inference

The large deviation is `get hits` and `put` can be due to mixing values for these operations when these were blocked by a remote operation and when these were not. There were no `get misses` because the entire page cache of the VM was flushed before the 2nd run when these readings were taken. Since the tmem cache at machine C is not evicted and

it contains all the pages of the file being remotified from B. Hence all `remotify_page` and `remotified_get_page` operations are successful and none fail. Because of lack of time the overheads for a failed remotification attempt was not explored. Lu and Du (2003) evaluates the performance of disk accesses across the various networks and using different technologies. Their work shows that the network latencies for 4KB sized data reads under different settings vary from 4ms — 25ms. If we assume the test machine to be operating in 1GHz and convert the values of the table 6.5 to depict the time taken by each operation, we get:

Operation	time in ms
<code>kvm_host_get_page (hit)</code>	0.77
<code>kvm_host_get_page (miss)</code>	0
<code>kvm_host_put_page</code>	0.34
<code>remotify_page (succ)</code>	1
<code>remotify_page (fail)</code>	0
<code>remotified_get_page (succ)</code>	0.92
<code>remotified_get_page (fail)</code>	0

Table 6.6: Table depicting the max possible overhead of tmem functions in terms of time in ms when tmem cache does both local & remote de-duplication. The frequency of CPU is assumed to be 1GHz

If the `VMExit` and `VMEntry` overheads in terms of CPU cycles are known then it can be determined what dominates among the network operations and the hypervisor exit/entry overheads. From the general information available it can be assumed the network operations dominates. But this needs to be verified.

# Chapter 7

## Extensions & Conclusions

Through this thesis the feasibility of letting a second chance page cache to access memory over network to exploit the presence of similar content in co-operating hypervisors was explored. At first sight this appears to be an alluring opportunity to improve memory utilization in data centers due the presence of large amounts of similar data and the fact that disk accesses there occur over network. This thesis confirms a hunch that over the network memory access has lower latencies than over the network disk access. Hence having a second chance page cache co-operatively access remote caches when under memory pressure certainly has its benefits. But more needs to be done towards empirically evaluating the overheads under realistic loads and settings to get deterministic results.

# References

- Article, LWN, 2010, “Transcendent memory in a nutshell [lwn.net],” [Online: accessed 18-October-2015], <https://lwn.net/Articles/454795/>
- Article, LWN, 2013, “In-kernel memory compression [lwn.net],” [Online: accessed 18-October-2015], <http://lwn.net/Articles/545244/>
- Fan, Li, Pei Cao, Jussara Almeida, and Andrei Z Broder, 2000, “Summary cache: a scalable wide-area web cache sharing protocol,” *IEEE/ACM Transactions on Networking (TON)* **8**, 281–293
- Hwang, Jinho, Ahsen J Uppal, Timothy Wood, and H Howie Huang, 2013, “Mortar: filling the gaps in data center memory,” in *Proceedings of the 4th annual Symposium on Cloud Computing (ACM)* p. 30
- Kloster, Jacob Faber, Jesper Kristensen, and Arne Mejlholm, 2007, “Determining the use of interdomain shareable pages using kernel introspection,” Department of Computer Science, Aalborg University
- Lu, Yingping, and David HC Du, 2003, “Performance study of iscsi-based storage subsystems,” *IEEE communications magazine* **41**, 76–82
- Mill, Bill, 2010, “Bloom filters by example,” [Online: accessed 18-October-2015], <http://billmill.org/bloomfilter-tutorial/>
- Miller, Konrad, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa, 2013, “Xlh: more effective memory deduplication scanners through cross-layer hints,” in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pp. 279–290

- Milós, Grzegorz, Derek G Murray, Steven Hand, and Michael A Fetterman, 2009, “Satori: Enlightened page sharing,” in *Proceedings of the 2009 conference on USENIX Annual technical conference*, pp. 1–1
- Mishra, Debadatta, and Purushottam Kulkarni, 2014, “Comparative analysis of page cache provisioning in virtualized environments,” in *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on (IEEE)* pp. 213–222
- Mishra, Debadatta, and Purushottam Kulkarni, 2015, “A survey of memory management techniques in virtualized systems,” Unpublished paper
- Oracle, 2010, “How to set up ramster,” [Online; accessed 18-October-2015], <https://oss.oracle.com/projects/tmem/dist/files/RAMster/HOWTO-v5-120214>
- Rachamalla, Shashank, Debahuti Mishra, and Parag Kulkarni, 2013, “Share-o-meter: An empirical analysis of ksm based memory sharing in virtualized systems,” in *High Performance Computing (HiPC), 2013 20th International Conference on (IEEE)* pp. 59–68
- Salomie, Tudor-Ioan, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone, 2013, “Application level ballooning for efficient server consolidation,” in *Proceedings of the 8th ACM European Conference on Computer Systems (ACM)* pp. 337–350
- Waldspurger, Carl A, 2002, “Memory resource management in vmware esx server,” *ACM SIGOPS Operating Systems Review* **36**, 181–194
- Wikipedia, 2016, “Bloom filter — wikipedia, the free encyclopedia,” [Online: accessed 7-August-2016], [https://en.wikipedia.org/wiki/Bloom\\_filter](https://en.wikipedia.org/wiki/Bloom_filter)
- Xen, 2015, “Cross reference: /xen/xen/common/tmem.c,” [Online: accessed 18-October-2015], <http://code.metager.de/source/xref/xen/xen/common/tmem.c>
- Zhao, Weiming, Zhenlin Wang, and Yingwei Luo, 2009, “Dynamic memory balancing for virtual machines,” *ACM SIGOPS Operating Systems Review* **43**, 37–47
- Zhou, Pin, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar, 2004, “Dynamic tracking of page miss ratio curve for memory management,” in *ACM SIGOPS Operating Systems Review*, Vol. 38 (ACM) pp. 177–188