Puma: Pooling Unused Memory in Virtual Machines for I/O intensive applications

Maxime Lorrillere, Julien Sopena, Sébastien Monnet and Pierre Sens
LIP6/CNRS/UPMC/INRIA, firstname.lastname@lip6.fr

ABSTRACT

With the advent of cloud architectures, virtualization has become a key mechanism. In clouds, virtual machines (VMs) offer both isolation and flexibility. This is the foundation of cloud elasticity, but it induces fragmentation of the physical resources, including memory. While each VM memory needs evolves during time, existing mechanisms used to dynamically adjust VMs memory are inefficient, and it is currently impossible to take benefit of the *unused* memory of VMs hosted by another host. In this paper we propose PUMA, a mechanism that improves I/O intensive applications performance by providing the ability for a VM to entrust clean page-cache pages to other VMs having unsused memory. By reusing the existing page-cache data structures, PUMA is very efficient to reclaim the memory lent to another VM. By being distributed, PUMA increases the memory consolidation at the scale of a data center. In our evaluations made with TPC-C, TPC-H, BLAST and Postmark, we show that PUMA can significantly boost the performance without impacting potential activity peaks on the lender.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management

General Terms

Design, Experimentation, Performance

Keywords

Linux, Virtualization, Cooperative caching

1 Introduction

Clouds extensively use virtualization which forms the basis to ensure flexibility, portability and isolation. However, the advent of virtualization tends to dramatically increase the amount of unused memory. A physical node is partitioned; its memory is split and distributed to multiple virtual machines (VMs). As it is very difficult to predict the amount of memory needed by an application, VMs memory is usually over-provisioned. The problem comes from the fact that the amount of VMs memory is defined *statically*; it is usually chosen among predefined configurations offered by cloud providers [Birke 2013]. On one physical host, the available memory is thus fragmented among the hosted VMs, leading to a global underuse of memory and inducing a huge extra cost. Some VMs may lack memory while others could use less without any (or only a little) performance degradation. These VMs can either be hosted on the same physical node or by different ones. In this context, providing the ability to pool *unused* memory at the scale of a data center would improve the global performance.

Several research works aim at improving memory usage, like deduplication [Miller 2012, Miłós 2009, Waldspurger 2002], or at offering memory flexibility like memory ballooning [Hines 2009, Schopp 2006, Waldspurger 2002] and hypervisor managed caches [Hwang 2014, Kim 2011]. However, these approaches exhibit severe limitations. Deduplication aims at optimizing only processes memory. Memory ballooning provides the ability to re-dimension VMs memory on a single host, but it was shown that it requires swapping to disk memory pages to handle changing workloads over time [Salomie 2013], which leads to a high response time. Finally, hypervisor managed caches such as Mortar [Hwang 2014] or XHive [Kim 2011] solve the problem of unused memory by providing a shared pool of memory maintained by the hypervisor, but assumes that: (*i*) VMs memory needs can be correctly predicted so that it is not necessary to dynamically resize it; or (*ii*) VMs memory can be dynamically resized when needed without a negative impact on performance.

In this paper, we propose PUMA, a system that is based on a remote caching mechanism that provides the ability to pool VMs memory at the scale of a data center. An important property while lending memory to another VM, is the ability to quickly retrieve the lent memory in case of need. Our approach aims at lending memory only for clean cache pages: in case of need, the VM which lent the memory can retrieve it easily. We use the inactive LRU of the page cache to store remote pages such that: (*i*) if local processes allocate memory the borrowed memory can be retrieved immediately; and (*ii*) if they need cache the remote pages have a lower than priority the local ones.

We show through extensive experimentations that our approach allows input/output (I/O) intensive applications to dynamically use free memory on remote VMs to boost their performance. Our evaluation on the benchmarks TPC-C, TPC-H, Postmark and BLAST using VMs localized either on a same or on different physical servers shows that:

- PUMA can bring a 12% speedup by borrowing only 500 MB, and up to 4 times faster using 6 GB of remote cache in the case of the TPC-C benchmark;
- PUMA can retrieve borrowed memory quickly, up to 10

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SYSTOR '15, May 26–28, Haifa, Israel
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

times faster than a ballooning-based approach;

- PUMA is resilient to latency variations: by monitoring the response time, it is able to decide if it should use the remote cache or fall back on disk accesses;
- in presence of sequential workloads, it is hard to improve performance, but PUMA does not degrade it thanks to a filter that prevent PUMA from using a remote cache in such cases.

The rest of this paper is organized as follows. First, Section 2 depicts motivating use cases for which PUMA can bring significant improvements. Section 3 gives some background information and details our remote cache design and implementation. Section 4 presents the experimental results obtained with PUMA, then Section 5 shows the benefits of PUMA on the use cases presented in Section 2. Section 6 presents related works and outlines the main differences with the PUMA's approach. Finally, Section 7 concludes the paper.

2 Motivating scenarios

To illustrate the benefit of PUMA we consider scenarios based on discussions we had with our industrial partners[1,2]. In the scenarios, a company runs a VM that can sometimes lent some unused memory. For example, a medium size IT company may need to run a VM with a database-based business activity and another one with revision control software such as Git or a continuous integration server to manage its projects source code. The business activity performs a lot of I/Os and may take advantage of a large cache while the continuous integration server has to allocate large amounts of memory during clone operations.

To emulate the business activity, we used the TPC-C benchmark configured to use 150 warehouses, which gives a dataset size of roughly 15 GB. This VM is configured to use 10 GB of memory. On the second VM, for sake of simplicity, we use a Git server that mirrors the Linux kernel repository to emulate a development activity. With such applications, each `git clone` command would temporary generate a memory peak load. This VM is configured to use 4 GB of memory to support such kind of peak loads.

Scenario 1. In the first configuration we consider, the company has bought a powerful multi-core node on which it runs the 2 VMs. Table 1 shows the response time of the `git clone` operations. The response time for the configuration with full isolation (*i.e.*, without any sharing mechanism) are given by the *baseline* line.

In this configuration, for which both VMs run on the same physical host, one may want to use the state-of-the art auto-ballooning approach [Capitulino 2013]. This approach provides the ability to significantly enhance the business activity latency by dynamically balancing memory of the VMs. However, as it was shown before [Hwang 2014, Salomie 2013], if the VM running the Git software suddenly needs a large amount of memory it will be slowed down. As shown in Table 1, 3 concurrent `git clone` operations take more than twice the time they use to take in the full isolated configuration[3]. The 4 GB VM has trouble to retrieve memory lent to the 10 GB VM. In PUMA, we detect the need of memory to quickly retrieve the pages that were lent to the TPC-C VM.

[1] `http://www.nuage-france.fr/`
[2] `http://en.oodrive.com/en/`
[3] This experiment is presented in details in Section 5.

	TPC-C	1 Git	3 Git
baseline	3.354s	215s	204s
Auto-ballooning	0.123s[a]	339s	447s
PUMA	1.186s	184s	251s

[a] The TPC-C VM was killed during the git clone due to out of memory.

Table 1: Automatic ballooning response time.

Scenario 2. We now consider another configuration in which a bigger company runs its own private cloud, or rents several nodes from a cloud provider. The company uses a 10 GB of memory VM to host its business activity, as in the previous case. It also has another 10 GB of memory VM that can be used as a spare VM if the first one fails. Meanwhile, this VM can let the 10 GB VM use its memory to extend its cache if needed. In that case, the memory ballooning approach is not an option anymore: it can only work for VMs running on a same physical host.

In this paper, we show how PUMA can enhance the performance of one VM by allowing it to borrow another VM's memory to extend its cache without slowing down the other VM memory allocation. We also demonstrate its ability to operate both locally (for VMs running on top a one physical node) and remotely (for VMs running on different hosts connected through a high speed network).

3 System Design

The basic principle of PUMA is to allow any cluster node to benefit of the memory of any other node to extend its cache. PUMA is designed using a client/server model, where a *client* node may use available memory on a *server* node. Any node can become a PUMA client or a PUMA server, and this can change among time depending on the memory needs. However, it does not make sense for a node acting as a client to also act as a server at the same time. This section first gives the necessary background, and then it describes PUMA's design, discusses the technical issues, and outlines its implementation.

3.1 Background

Virtualization. Virtualization enables multiplexing the computing resources of a physical server. A *hypervisor* [Barham 2003, Kivity 2007, Velte 2010, Waldspurger 2002] runs on a physical *host* and is responsible for managing Virtual Machines (VMs). VMs can run *guest* operating systems without any modifications, *i.e.*, in *full virtualization*, but it involves a performance degradation. *Paravirtualization* is a virtualization technique that presents specific interfaces to the virtual machines in order to reduce the virtualization overhead. *Paravirtualization* is especially useful to improve (I/O) performance.

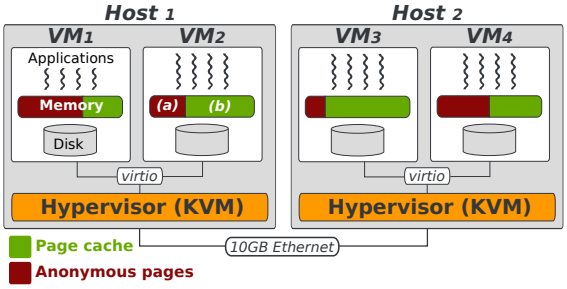


Figure 1: Virtualization and system page cache.

Figure 1 shows a typical virtualized configuration with 2 hosts. Each host runs two guest operating systems on top of a hypervisor (*e.g.* KVM), with a paravirtualized network (*e.g. virtio*). Using a paravirtualized network improves the overall network performance, but it also allows guests running on a same host to communicate through a high performance paravirtualized network.

Page cache. To enhance file access performance, operating systems keep the data read from disk in a cache called a *page cache*. In the Linux kernel, these pages are stored into a per-inode radix tree. Page cache pages are often called *file backed pages* because there always exists a corresponding image on disk. When the system load is low, most of the memory is filled by pages from the page cache. When the system load increases, the memory is filled mostly by processes pages, the page cache is thus shrunken to make room for the active processes. The process pages are called *anonymous pages*, *i.e.*, pages that are not file backed. In Figure 1, we show the anonymous pages (a) and file backed pages (b) for each VM.

Page frame reclaiming. When there is a need for memory, for instance when a process tries to allocate memory or when the *kswapd* kernel threads wake up, the *Page Frame Reclaiming Algorithm* (PFRA) of the Linux kernel frees some memory pages. To do so, both anonymous and file backed pages are linked together into two LRU lists. The PFRA chooses *victim pages* to evict from memory. Most of the time, *clean* file-backed pages are evicted if possible because doing so is inexpensive: they have an identical version on disk and thus they just have to be freed. If page cache pages are *dirty* (*i.e.*, modified), they must first be written to disk. When anonymous pages are evicted, they also have to be written to disk into the *swap* space.

3.2 Design Goals

PUMA's main goal is to pool VMs unused memory for the benefit of other VMs having I/O intensive workloads. We chose to handle only *clean* file-backed pages through PUMA for the following reasons:

- If processes running on the server need to allocate memory, the *clean* file-backed pages of the client can be removed without any synchronization.
- Writes are generally non-blocking, because writing dirty pages to disk can be deferred, thus there is no performance improvement to expect in handling writes into the cache. In contrast, reading a block from disk is blocking and as slow as disk latency.
- Managing dirty pages into a cooperative cache is complex because consistency issues have to be handled in case of failure. Having such an overhead without performance increase is useless.

As we choose to handle only *clean* file-backed pages, fault-tolerance is straightforward since it is only necessary to detect the failure of a PUMA node. If a node crashes, an up-to-date version of the pages remains available from the disk. Hence, we consider that fault tolerance is out of the scope of this paper.

Finally, PUMA should be designed such that it can be used with any file system, block device or hypervisor.

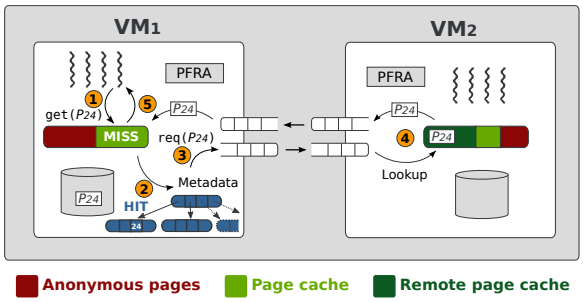


Figure 2: **get()** fetches a page from the remote cache.

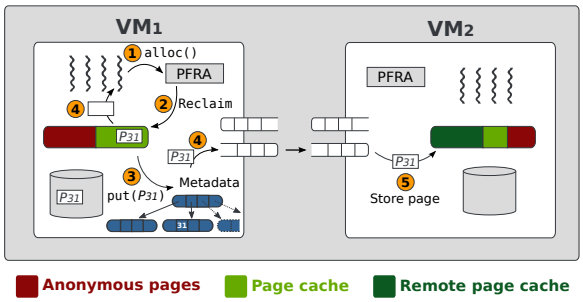


Figure 3: **put()** stores a page to the remote cache.

3.3 Puma Design

We want our cooperative caching mechanism to be *general*, usable with block devices, but also by distributed file systems. A straightforward method to build a low-level cooperative cache is to build a virtual block device on top of a real one. The virtual block device can thus catch every attempt to read from the disk, *i.e.*, every miss from the page cache. It can then make a lookup into the cooperative cache to try to find the missing data. However, even if this solution is simple and elegant, it severely limits the scope of the cache to block devices only, and thus prevents distributed file systems from using it.

We prefer to adopt a more general approach. To do so, we have to catch directly misses and evictions from the local native page cache as shown in Figures 2 and 3 where VM_1 and VM_2 act as a client and a server respectively. Each miss from the page cache can lead to a *get* operation while evictions lead to *put* operations.

Get operation. In case of a miss on a clean page in the local cache (step 1 of Figure 2), PUMA checks in its local metadata catalog if this page has been sent to the server (2). This metadata catalog, maintained by the client, stores the ids of the pages that have been sent to the server during a *put* operation in order to avoid sending useless *get* requests to the server. Thus PUMA sends a request to the server (3), only when the page id is present in the metadata catalog. Then, the server sends back the page (4).

Put operation. When a victim page is chosen by the PFRA in order to free some memory (steps 1 and 2 of Figure 3) it may induce a *put* operation (3) if PUMA decides it is worth if (according to the mechanisms described in sections 3.3.2 and 3.3.3). Then, PUMA copies the page to a buffer while the freed page can be given back (4). Finally, the page is sent to the server to store it (5).

3.3.1 Dealing with the memory

While implementing the *get* and *put* operations, it is necessary to take memory contention into account. Indeed, the put operation is called when the PFRA tries to solve a lack of free memory. However the put operation needs memory, at least to allocate the necessary data structures to send the page, such as network buffers or local metadata. The put operation should avoid allocating memory; this could lead to make the PFRA trying to free some more memory pages, leading to another call to the put operation, and so on. In practice, this would end in a put failure: the evicted pages will not be sent to the remote cache (however, this remains correct since PUMA only puts clean, file-backed pages in the remote cache).

To take into account the memory contention and to increase the global performance of caching, PUMA is based on the following principles.

Preallocation of request buffers. We strive to send the pages to the remote cache to give more chance for a remote cache hit. To increase the probability that a *put* will succeed, all the memory needed to handle requests is taken from a preallocated pool of memory. Thanks to this, we do not add pressure on the PFRA.

Aggregation of read-ahead get requests. To avoid blocking processes each time a page is read, we aggregate all the pages of the read-ahead window together within a single get request sent to the server. Therefore, we have only one blocking request operation for all these pages. Thanks to this approach we benefit from the existing read-ahead algorithm.

Aggregation of put requests. The memory reclaim path generally chooses tens of victim pages to reduce the number of calls to the PFRA. Sending tens of messages is not very efficient and consumes the memory we are trying to release. To avoid the creation of many small messages, we use a per-process message buffer to merge messages together.

3.3.2 Dealing with response time

PUMA relies on short response times. To avoid a performance drop PUMA monitors the latency and, in case of high latency, it stops using the server and falls back on disk accesses.

To monitor the latency, PUMA nodes periodically exchange *ping* messages. They compute both a short-term L_{short} (the last 15 seconds) and a long term moving-average latency L_{long}. The first one is used to detect a latency peak, and the second one to measure the average latency.

When one of these averages reaches a certain threshold, the PUMA node stops sending *put* or *get* requests to the other node and falls back on disk accesses. When the latency gets below another threshold, the PUMA node starts sending *put* or *get* messages again. We use different thresholds to provide some hysteresis. We provide an analysis of this mechanism in Section 4.5.

3.3.3 Dealing with sequential workloads

Disks usually support a bandwidth of hundreds of MB/s, which might be higher than the network bandwidth available to PUMA. Moreover, such accesses are generally prefetched, which means that disk access latency is amortized. In such cases, PUMA might slow down an application which does sequential I/Os. To avoid a performance drop with this kind of I/O pattern, we introduced an optional *filter* to PUMA. The filter aims to avoid using the remote cache for sequential accesses and to focus only on random accesses.

When this option is enabled, PUMA detects sequential accesses from disk, and tags the corresponding pages such that, when they are evicted from the local cache PUMA does not send them to the remote cache. The benefit of the *filter* option is twofold:

- the corresponding pages are not sent to the cache, this means that we avoid the overhead due to a large *put*;
- on a second access on these pages, they are not present into the remote cache and can be retrieved efficiently from disk; hence PUMA will not slow down the application by fetching them from a remote node.

The *filter* option is analyzed in details in sections 4.3 and 4.4.1.

3.3.4 Caching Strategies

In this section, we present two different caching strategies that we have implemented for PUMA.

Exclusive. With the exclusive strategy, when a client requests a page, it is removed from the server's memory. This strategy does not require maintaining different copies of a same page. Moreover, since the remote cache is used *in addition* to the system one, the total available cache size is the sum of the size of the local cache and the size of the remote one. However, this strategy will make a client send a same page to the server many times, particularly in workloads with frequent read requests.

Non-Inclusive. The non-inclusive strategy aims at reducing client and network loads in read dominant workloads. In our implementation, the server keeps pages in its memory even when they are sent back to a client. Thus, hot pages remain in the server memory. This may prevent the client from needing to send this page again to the server if it is later chosen as a victim page. In contrast to a *strictly-inclusive* caching strategy, a *non-inclusive* [Jaleel 2010, Zahran 2007] caching strategy is a strategy where the inclusion property is not enforced. Thus, the total cache size available is closer to the exclusive strategy, while with a *strictly-inclusive* strategy it would be the *max* between the size of the local cache and the remote one.

3.4 Implementation Details

We implemented PUMA in the Linux 3.15.10 kernel. Most of our implementation is ~8,000 lines of code inside a kernel module. A few core kernel changes are necessary: ~150 lines to the memory management subsystem and ~50 lines to the virtual file system layer.

3.4.1 Metadata management

A client keeps track of meta-information sent to the server in a radix tree. Clients maintain a small amount of metadata for each page cached by the server to handle consistency and to be able to check locally if a given page is stored by the server. This metadata includes some bits (*i.e.*, *present*, *busy*) to know whether or not the page is into the cache or if a page is being sent. More bits might be used to locate the PUMA node where the page was previously sent.

Overall, for each page (4 kB) stored in the remote cache, a client needs to keep only a few bits embedded into a single 64 bits integer, which means that we only need 2 MB of memory (amortized) on client side to manage 1 GB of memory in the remote cache. Moreover, a client has always the possibility to reclaim the memory used by the metadata; in this case, it invalidates the corresponding pages on the remote cache.

3.4.2 Server side implementation

A server offers its free memory to the remote cache. It handles requests from a client. Pages on a server are stored into an in-memory tree and linked together into the system LRU lists and are accounted as *inactive page cache pages*. Thus, remote cache pages reclaiming works in the same way as local pages. However, as remote cache pages are added to the inactive LRU list, if a process on the server VM needs memory, either for caching I/Os or to allocate anonymous pages, remote cache pages will get evicted before local pages.

3.4.3 Non-inclusive cache consistency

Clients only act on clean pages[4] and we have no means to detect if a given page has been modified since the last time PUMA fetched it from the server. This means that the server may have an old version of a modified page, in which case the client has to send it again to the server, even if it detects that the page is already stored by the server. To solve this problem, we chose to add a `dirtied` flag to each page, in addition to the existing `dirty` bit, in order to catch every time a page is set dirty. We can then check this `dirtied` flag when the page is evicted from the page cache.

Moreover, due to buffering a dirtied (*i.e.*, updated) page may be queued into the message buffer while a process is trying to get that page (as PUMA buffers pages to be sent to the server). This may lead to a scenario in which a request for the page reaches the server, potentially storing an old version of the page, before the page itself. We solve this race condition by adding a *busy* synchronization bit on the metadata.

4 Evaluation

In this section, our objective is to evaluate how PUMA behaves in various situations. First, we describe the experiment setup; then we evaluate the raw performance of PUMA with micro and applicative benchmarks and we analyze the benefit of both caching strategies. Then, we compare the performance of PUMA with SSD caching, and we analyze how PUMA is able to dynamically reclaim memory when needed.

4.1 Experiment setup

4.1.1 Setup

All the experiments were run on the high-end *Paranoia* cluster from the Grid'5000 platform [Bolze 2006], where each node is a 2×8 cores Intel Xeon E5-2660v2, with 128 GB of memory and a 10 Gb/s Ethernet card. These nodes also have 5 600 GB SAS disk, that we configured in RAID0.

We deployed the benchmarks on VMs under the Linux KVM [Kivity 2007] hypervisor that we configured according to best practices [IBM 2010]. Each VM uses 2 virtual CPUs with the ext4 file system, and all I/O are done through the *Virtio* [Russell 2008] paravitualization framework. Each experiment is done on a freshly booted VM after a warm-up phase long enough to fill the system page cache and the remote cache. We run each experiment 10 times and then we compute the average and the confidence interval using the student's t-distribution with a 95% confidence level. We always observed a small standard deviation.

4.1.2 Workloads

We used the following benchmarks in the experiments:

Random reads workload: we use the *random reads* workload from the *Filebench*[5] benchmark. This workload starts one thread reading from a single file at non-aligned random offsets. We configured it to use a 4 GB file with a read size of 4 kB. We measure the number of completed I/O per second.

Sequential reads workload: we use the *sequential reads* workload from the *Filebench*[5] benchmark. This workload reads a single 4 GB file multiple times from the beginning, and reports the resulting bandwidth (MB/s).

Scientific workload: *BLAST* [Altschul 1990] is a bioinformatics tool used to find regions of similarity between a query sequence and a database of sequences. Basically, BLAST scans the database to find sub-sequences which are similar to the query. A large part of the I/O pattern generated by BLAST is sequential. For our experiments, we used the *patnt* database, which is roughly 3 GB in size. We extracted 5 sequences of 600 characters from this database, and we measure the runtime needed to find similar sequences in the database.

Write-intensive workload: *Postmark* [Katcher 1997] is a benchmark which measures the performance of file systems over a workload composed of many small files, typical of e-mail services. This workload can generate a mix of data and metadata operations, and is thus composed of many small writes to the file system. Postmark defines a transaction as a read or an append to an existing file, followed by a file creation or deletion, and reports the number of completed transactions per second. We use an unfriendly configuration of Postmark where more than 80% of I/Os are writes. We configured Postmark to generate $20,000$ transactions on $25,000$ files. We chose file size ranging from 512 bytes to 64 kB to generate a working set of around 3.5 GB. Files are distributed among 10 subdirectories. I/Os are unbuffered and done with 4 kB I/O size. We configured postmark to generate transactions with the same ratio of reads over append, and with the same ratio of creations over deletions.

Database workloads: we use *TPC-H*[6] and *TPC-C*[6] as database benchmarks. Both of them were run on the version 9.3.1 of the PostgreSQL database server.

The first one defines a set of 22 complex business-oriented ad hoc queries and concurrent data modifications. Most of the workload is read-only. It models queries executed in large scale enterprise that examine large volume of data to give answers to critical business questions. We configured it with a scale factor of 3, giving a data set size of around 3 GB. TPC-H reports a throughput in terms of number of completed queries per hour.

TPC-C simulates a complete computing environment of an online transaction processing marker. We configured it to use 40 warehouses, giving a data set size of roughly 4 GB. TPC-C measures the number of "New-Order" transactions executed per minute and reports their response time. In our evaluation, we only consider the response time (90th percentile) of this transaction.

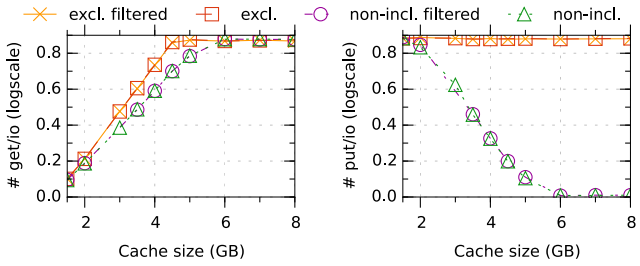
4.1.3 Metrics

In all the experiments we used the previously described benchmarks to evaluate PUMA. To this end, we vary the size of the available PUMA cache and we compute their *speedup* relative to a 1 GB VM with no additional cache. Thus, we

[4] Dirty pages are written to disk before put() is called.

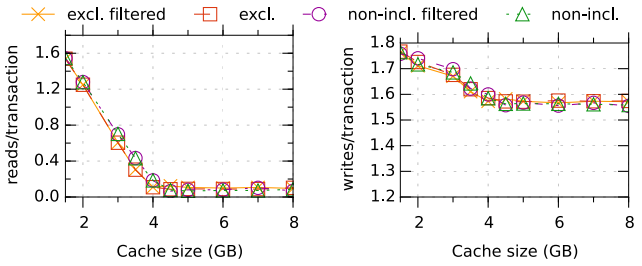
[5] `http://filebench.sourceforge.net`

[6] `http://www.tpc.org`



(a) Number of gets (b) Number of puts

Figure 4: Number of gets/puts for the random read benchmark.



(a) Average read/transaction (b) Average writes/transaction

Figure 6: Accesses to block device with Postmark.

used 2 VMs for PUMA: VM_1, where the I/O intensive benchmarks runs, which has 1 GB of memory; and VM_2, which offers its free memory to VM_1 through PUMA.

4.2 Random reads and caching strategies

In this section, we study the random reads benchmark performance and show PUMA's exclusive and non-inclusive caching strategies behavior.

With a random read access pattern, the hit ratio grows linearly with the available cache. However, as the Figure 5a shows, the performance of the cache depends on the access time of the slow device (in case of a miss). For instance, 1% of misses is enough to drop the performance. This is why we observe a threshold effect when the entire working set fits in the cache.

To analyze the benefit of each caching strategies, we study the number of pages sent to (*put*) and fetched from (*get*) the remote cache of this benchmark.

Figure 4a shows the number of *get* per I/O of the random read benchmark. The average number of pages get from the remote cache increases linearly with the memory available for caching. We see that the exclusive strategy does more *get* requests (*i.e.* remote hits) than the non-inclusive one until the whole working set fits in the cache. This is consistent with the performance results in Figure 5a, where the exclusive strategy is better than the non-inclusive one because the total cache size available is higher.

Figure 4b shows the number of puts per I/O for the random read benchmark. With a non-inclusive strategy, we observe that the number of pages sent to the remote cache decreases as the cache size increases, while it remains constant with the exclusive strategy. This is because this benchmark is read-only, which allows the non-inclusive strategy to avoid sending back pages which are already present to the remote cache. However, as the Figure 5a shows, the exclusive strategy performs better than the non-inclusive one until the entire working set fits in the cache, which illustrates the overhead of PUMA's *put* operation.

4.3 Filtering sequential I/O

In this section, we analyze the performance of the sequential reads benchmarks, and we show that with the *filter* option, PUMA is able to detect sequential patterns to avoid a negative impact on the performance.

As the nodes we are using high performance SAS disks configured in RAID0, it is hard to improve the performance of a sequential I/O pattern. This is confirmed by the performance results of the sequential reads benchmark presented in Figure 5b. First, we can see that PUMA is thrashing until the whole file can fit in the cache, thus we always pay the price without any benefit. Next, when the cache is large enough, the exclusive strategy does not help because half of the network bandwidth is used to send the pages to PUMA, while with the non-inclusive strategy the pages stay in the cache.

However, when the *filter* option is enabled, sequential accesses are detected and go directly to the disk so that PUMA does not slow down the accesses.

4.4 Applicative benchmarks performance

4.4.1 BLAST: *partially sequential workload*

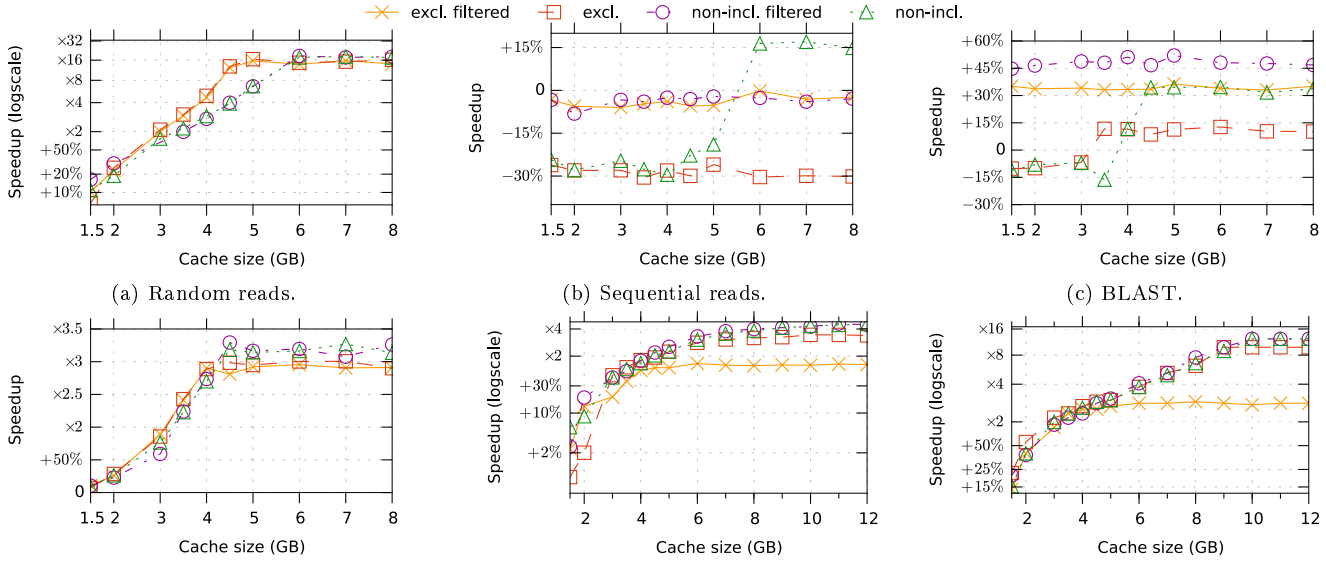
As we saw for the sequential reads workload, it is difficult to improve I/O performance for a sequential pattern due to cache thrashing and amortized disk latency. As expected, the speedup of *BLAST* presented in the Figure 5c shows that PUMA with no *filter* option degrades the performance of the application if the database does not fit in cache. However, when the cache is large enough, PUMA is able to improve the performance of *BLAST* up to +30%.

Surprisingly, if we enable PUMA's *filter* option, PUMA is able to improve the performance up to +45%. This is due to the fact that, as we explained in section 3.3.3, the *filter* option has a double benefit: (*i*) the application is not slowed down due to the higher bandwidth of the storage device compared to the network bandwidth, and (*ii*) we give more room to quickly send random *get* requests by not overloading PUMA's message queues.

4.4.2 *Write-intensive workload*

Postmark is designed to simulate the workload of applications like e-mail services that makes an intensive use of writes, which could be a worst case for a read-only cache like PUMA. However, as the Figure 5d shows, PUMA can still improve the performance even if it is not designed to handle dirty pages: a small amount of cache is enough to get around 10% of performance improvement. We can improve the performance up to 3 times compared to the baseline when all the dataset fits in cache.

To understand how PUMA can improve write intensive workloads, we report the number of I/O sent to the block device per transaction executed into the Figure 6. As expected, we observe that PUMA reduces read accesses (Figure 6a), however we can see that it also reduces the number of writes (Figure 6b). This phenomenon is due to the fact that in this benchmark, writes are buffered, thus they are sent to the block device if the *PFRA* needs to flush them to free memory. However, as it takes time to write data, the *PFRA* has to reclaim clean pages first. Thus, by increasing the cache size, more writes can be delayed, which reduces the I/O load. With PUMA, clean pages are sent to the PUMA node, which gives more memory to delay the writing of dirty pages.



(a) Random reads. (b) Sequential reads. (c) BLAST.

(d) Postmark. (e) TPC-H. (f) TPC-C.

Figure 5: Speedup relative to a 1 GB VM obtained with various benchmarks and applications: (a) 4 kB random reads, (b) sequential reads, (c) BLAST, (d) Postmark, (e) TPC-H and (f) TPC-C.

4.4.3 *Database workload*

The TPC-H and TPC-C benchmarks are highly concurrent I/O workloads that are very dependent on cache size. As the Figures 5e and 5f show, PUMA is able to improve the performance of both of them even with a small cache size: at least +5% for TPC-H and +12% for TPC-C. The concurrency of these benchmarks can be observed by the exclusive caching strategy with the *filter* option enabled that reaches its limit very quickly. This is due to the combination of two factors:

- Most of the I/Os are a mix of random accesses and medium sized sequences. With the non-inclusive strategy (with *filter*), once a page is accessed randomly, it stays in the cache even when it is accessed a second time sequentially. In the case of the exclusive strategy (with *filter*), pages are removed from the remote cache when accessed.
- Sequential accesses are not as fast as in the sequential read benchmark or the *BLAST* application because multiple concurrent streams generates concurrent sequential I/Os that involve a lot of disk seeks. Hence, such I/Os are more subject to a performance improvement.

4.5 Resilience to network latency

In this section, we show that network latency is a critical issue and that PUMA is able to control itself to avoid a negative impact on applications performance. To show the problem, we used Netem [Hemminger 2005] to inject network latency between PUMA nodes, and we measure the speedup of the benchmarks with PUMA relative to a 1 GB VM without PUMA (as in previous sections). Figure 7a shows the results of these experiments.

As we can expect, most of the overall speedup of the applications with PUMA decreases as the network latency increases. Benchmarks that are the most I/O intensive are slowed down compared to the baseline: Postmark performance is reduced with an additional network latency of 500μs, 1ms is enough to slow down the sequential read benchmark, and TPC-C response time skyrockets off the chart with more than 1.5ms.

Figure 7b shows the results of these experiments with PUMA latency management mechanism enabled. We configured (empirically) L_{short} to $[1.5ms, 40ms]$ and L_{long} to $[1ms, 1.5ms]$. Overall, we observe that this mechanism helps PUMA to not slow down applications performance in case of high latency. The only notable exception is Postmark with 500μ of network latency, where in this case T_{short} and T_{long} need to be more aggressive.

4.6 Comparison with SSD caching

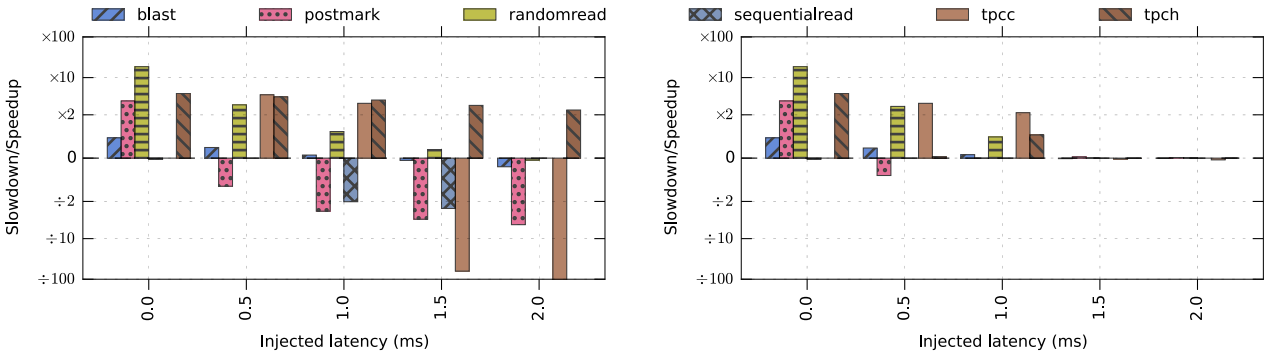
In this section, we compare PUMA's performance against SSD caching.

For these experiments, we used a Samsung 840 Pro 128GB with the *dm-cache* module from Linux kernel. dm-cache is a *device-mapper* target which creates a virtual block device on top of both a real device (HDD) and a *cache* device (SSD). Basically, when an I/O is issued to the virtual block device, dm-cache first tries to execute this I/O on the cache device. We configured dm-cache inside a VM to use a 5 GB cache device backed by the SSD, *i.e.*, the working set of our benchmarks fits in the cache. We also run PUMA with a non-inclusive strategy and 5 GB of remote cache.

The results of these experiments are presented in table 2. These results show that under random read workloads, PUMA performs much better than a SSD cache, while we expected to have more or less the same result from both. The reason is that the virtualization overhead is a known issue [Har'El 2013], especially for I/O: while we got around 150μs latency between VMs for PUMA, which is pretty high, we measured more than 200μs of access latency to the SSD from the VM.

With BLAST, since it has a sequential pattern, the SSD is not impacted by the latency, and the SSD cache is roughly as fast as PUMA.

TPC-H generates a mixed sequential/random workload with writes. With this benchmark, PUMA is still faster than the SSD cache, but since we do not handle writes and writes



(a) PUMA without latency management.

(b) PUMA with latency management.

Figure 7: Speedup and slowdown (logarithmic scales) of benchmarks with latency injection between VMs, without (a) and with (b) latency control support.

	Random read (IO/s)	Postmark (T/s)	BLAST (s)	TPC-H (throughput)
baseline	166	64.7	42.3	248
PUMA	**6310**	100.5	**17**	**791**
dm-cache	3250	**695.2**	21.3	566

Table 2: Performance comparison between SSD caching and PUMA, both with 1 GB of local memory and 5 GB of additional cache.

to an SSD are much faster than writes to a magnetic disk, the gap between the SSD cache and PUMA is reduced.

Finally, with Postmark, as we could expect the SSD cache is much faster than using PUMA since we do not handle writes, and thus we have to write them to a slow HDD. It shows us the cost of writes into the HDD: dirty pages are all eventually written to the slow HDD, while with an SSD cache, which is persistent, they can only be written to the SSD. To summarize, SSD caching performs well under write workloads and it's even better to commit writes to an SSD than to delay them in memory.

4.7 Dynamic memory balancing

In this section, we evaluate PUMA's dynamic memory management, which is detailed in Section 3.4.2, then we measure its impact with varying workloads compared to a memory ballooning approach.

We deployed the random read benchmark described in Section 4.1 with a 64 KB block size on a VM with 1 GB of memory. Another VM with 4.5 GB is available for caching with PUMA's non-inclusive strategy. On this second VM, we inject a memory workload after 10min, which consists on allocating small chunks of memory each second, emulating an application need for anonymous memory. The size of the chunks is chosen such that all the memory of the VM is allocated in 10min. We write into each allocated memory chunk to ensure that the pages are really mapped into the VM. Then, this benchmark sleeps for ten minutes before freeing the allocated memory in a few more minutes.

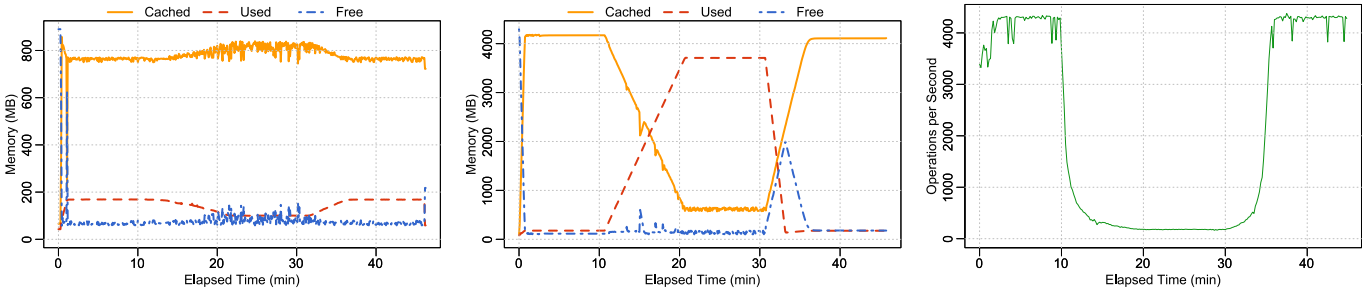
We measure the memory consumption and the cache activity for both VMs. On the first VM we also measure random reads performance, and on the second we study the latency of each memory allocation.

Figure 8a shows the memory usage of the client VM running the random read benchmark. We observe that when the server reduces its cache size, the memory used for local caching increases. This is due to the fact that there is less metadata to store on client side (the metadata being used to recall which pages are stored remotely) freeing some memory that can be used for caching. Before starting the memory allocation workload on the server side, the amount of local memory used for caching is stable. However, when the memory workload starts (at 10'), this amount of memory is more variable because the remote cache is reducing the available memory for remote-caching, and thus the entire working set (4 GB) does not fit in cache and the client, using the non-inclusive strategy, has to send pages to the remote cache.

Figure 8b shows the memory usage of the VM which offers the PUMA remote cache. At 10', it starts the memory workload and, dynamically, reduces the remote cache size to allow pages of the memory intensive process to be allocated. When this process frees its memory, the memory is immediately available again for the benefit of the other VM. Note that, as explained in section 3.4.2, PUMA links remote cache pages directly into the system LRU to improve memory reclaiming and that they are accounted as *cache pages*. This is why the *cached memory* curve is high when the VM stores remote cache pages, while it drops when the memory workload starts.

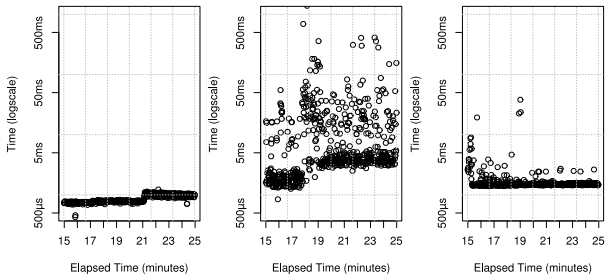
Figure 8c shows the performance of the random read workload. Before the PUMA remote cache node starts the memory workload, all the data set fits in the cache (local+remote) and the performance is at its maximum. When the memory workload starts, the performance quickly collapses, because, as we saw in Section 4.2, only a few disk accesses are enough to slow down such kind of workload. When the memory workloads frees the memory on the remote cache node, the remote cache is progressively filled, which increases the hit ratio and the performance.

We report on the Figure 9 the latency of each memory allocation for a configuration using KVM memory autoballooning and for PUMA. We also report these values for a single VM with 4.5 GB of memory (*ideal*). With this configuration, memory allocations take less than 1ms, with a standard deviation of 0.1.



(a) Client side memory (b) Server side memory (c) 64KB random read workload with varying memory load on server side.

Figure 8: Memory usage with dynamic memory management and varying memory load on server side.



(a) Ideal (b) Ballooning (c) Puma

Figure 9: Memory allocation latencies.

With dynamic memory ballooning, memory allocations take 20ms on average, with a standard deviation of 76 (some values, not represented in this figure, are above 1s). This is because the VM running the random read benchmark took all of the memory for its own cache, and it is not possible to deflate the balloon of the VM running the memory workload without swapping. This is due to the *semantic gap* between the hypervisor and the VM: the host does not know that the memory is used for caching and can be reclaimed.

With Puma, the logic is embedded inside the VMs, and we are able to get back the memory used for caching without the need of the hypervisor. In average, we measure 1.8ms of latency for memory allocations, with a standard deviation of 2.2.

5 Back to the scenarios

In Section 2 we presented 2 scenarios in which existing solutions are either non-efficient (locally) or non-existent (remotely). In this section, we show that Puma is able to dynamically use free memory of other VMs, hosted locally or on another host, to improve the performance of I/O intensive applications, without impacting potential activity peaks.

5.1 Co-localized VMs

In our first case study presented in Section 2, we show how Puma can be used to improve the performance of consolidated applications into a single server. The results of these experiments are presented in Table 1. As we can see, automatic ballooning helps to improve the performance of the TPC-C response time. However, it cannot return the memory to the Git VM because of the semantic gap: most of the memory used by TPC-C is for caching purpose and could be reclaimed with almost no overhead. This leads to a huge performance drop of the Git VM (219%). With Puma, the

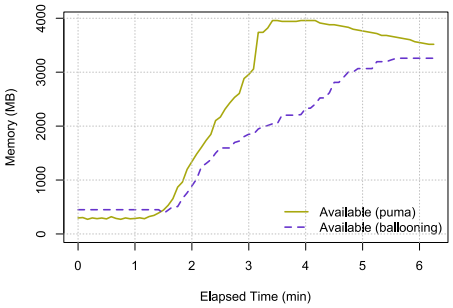


Figure 10: Available memory on the git server with Puma and auto-ballooning. While it is idle, all the memory of the VM is used for the benefit of the TPC-C VM. On activity, Puma quickly reclaims the memory lend to the other VM.

performance of the TPC-C VM is still improved (282%), but not as much as automatic ballooning. The performance of the Git VM is still impacted (23%), but it is much more acceptable than what we observe using the auto-ballooning approach.

To explain why automatic ballooning fails, and how Puma is able to succeed, we report the amount of memory available to the Git VM in both cases in Figure 10. With Puma, we represent the amount of free memory available into the VM, minus hosted remote cache pages, while we report the amount of available memory (*i.e.* memory hot-plug) in the auto-ballooning. For the first part of these curves, almost no memory is available: free memory is either used for remote cache pages (Puma) or directly given to the other VM (auto-ballooning). In the second part of these curves (1'30"), the `git clone` has started and the Git server has to allocate memory. In the case of Puma, the memory is quickly reclaimed and the application can progress. Using the auto-ballooning approach the host is not able to reclaim the memory used by the TPC-C VM, and it has to swap to give the memory needed by the Git VM, which is a very slow process. These results are consistent with what we have observed in the previous experiments (Figures 8b and 9). When the activity of the Git VM stops, Puma starts offering again free memory to the TPC-C VM.

5.2 Distributed setup

As described in Section 2, the second scenario represents a bigger company which has multiple physical servers in a data center, interconnected with a high performance network, such as 10 GB Ethernet. On the first node, we deploy a VM with 10 GB of memory and we run a database

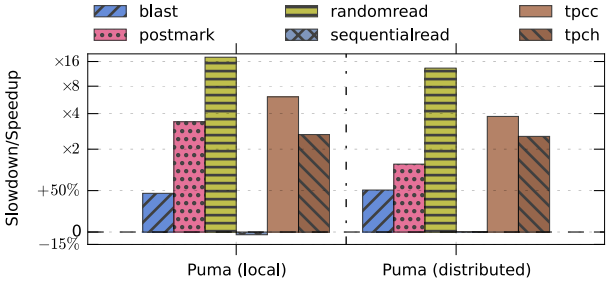


Figure 11: PUMA speedup with VMs on different nodes.

benchmark as in the previous section. On the second node, we deploy a *spare* VM with 10 GB of memory, which can be used in case of a failure of the first VM. Thus, until there is a failure, the spare VM can offer its unused free memory to the main VM, by using PUMA and the 10 GB Ethernet link. Obviously, we cannot reproduce these experiments with the automatic ballooning approach, as such solutions do not work among multiple nodes.

The results of this experiment are presented in Figure 11. We also presents the results for the TPC-H benchmark with $10 + 10$ GB of memory, and we reproduce the results of the other benchmarks that we presented in Section 4.1 in their 8 GB configuration. Surprisingly, overall performance in a distributed setup is close to the performance in a local setup. However, as we already explained I/O virtualization is still slow, and the use of a high performance network only add tens of μs of latency.

6 Related Work

Many research efforts have considered the problem of how to improve VM memory usage in clouds. Most of these approaches provide the ability to share memory among multiple physical machines, but only few of them focus on how to optimize the usage of the "unused" memory (*i.e.*, file backed pages). In this section, we briefly present these approaches and explain how our solution differs.

Memory ballooning. Memory *ballooning* [Barham 2003, Schopp 2006, Waldspurger 2002] is a memory balancing technique, it allows a hypervisor to dynamically resize the physical memory it gives to the hosted VMs. To do so, it instructs a guest to *inflate* a balloon, which pins memory pages within the guest. To increase the memory of another guest, the hypervisor asks the inflated guest to *deflate* its balloon. The major drawback is that, due to the *semantic gap*, the hypervisor does not know for what the memory given to the VMs is used, hence it is not able to reclaim memory even if it is used for caching. This solution presents other drawbacks: (*i*) inflating/deflating balloons may incur latency, and (*ii*) it only operates for VMs running on top of a single physical host. In contrast, PUMA is able to handle varying workloads and can be deployed on VMs hosted on different hosts.

Page sharing (deduplication). Memory ballooning is often used with page sharing among VMs. Transparent page sharing [Waldspurger 2002] is a widely used technique to reduce VM memory usage and to overcommit memory. Basically, the hypervisor periodically scans the physical memory of each guest and when identical pages are detected, they are shared so that the VMs access the same pages. The main drawback is that scanning the memory consumes CPU and memory bandwidth. Miłoś *et al.* proposed Satori [Miłós 2009], a modification to the Xen hypervisor that uses *enlightenments* from the guest to help the hypervisor to scan pages more efficiently and to detect short-lived sharing opportunities. However, even if these solutions optimize the memory usage by avoiding to store multiple times the same data, and offer good results, there is still room for improvement. They cannot solve the sizing problem: they can *compress* the memory allocated to a VM, but what if the VM needs more memory? These approaches appear to be complementary to ours.

Cooperative caching. Cooperative caching [Dahlin 1994] uses participating clients' memory to extend the *local* cache of other clients.

Some approaches take benefit of the virtualization to provide a transparent cooperative cache between VMs that may be used to pool the unused memory of the VMs. XHive [Kim 2011] is, in some ways, a paravirtualized cooperative cache. However, while their cooperative caching approach gives good performance, it requires to make complex modifications both to the guest OS and to the hypervisor. Moreover, like the ballooning approach, it cannot be used to pool the memory of VMs hosted on different hosts.

Hwang *et al.* proposed Mortar [Hwang 2014], a framework used to pool spare memory of Xen virtual machines to provide volatile data cache managed by the hypervisor. In this approach, a predefined amount of memory is allocated by the hypervisor and can be used on-demand by the VMs. In contrast, PUMA is able to manage the unused, but allocated, memory of the VMs, while Mortar manages a preexisting pool of free memory, i.e. VMs have to be resized in order to give their unused memory to the hypervisor. Moreover, PUMA does not rely on hypervisor modifications, allowing it to be deployed regardless of the hypervisor.

Zcache (formerly RAMster) [Magenheimer 2009] is a compressed in-memory page cache that periodically sends compressed pages to an other host. Zcache works for *anonymous* pages as well as clean *page cache* pages; however previous studies [Magenheimer 2012] show that most of the benefit of Zcache comes from *anonymous* pages (*i.e.* swap) in memory intensive workloads such as multithreaded kernel compilation. Moreover, the compressed cache tends to grow so fast under I/O intensive workloads that it is not able to send page cache pages to a remote host.

7 Conclusion

In cloud architectures, the extensive use of virtualization through VMs leads to a fragmentation of the memory. To tackle this problem, we propose PUMA, a mechanism providing the ability to pool unused memory. As a VM should be able to retrieve quickly the memory it has lent to another VM, we based our approach on lending memory only to store clean file-baked pages that can be removed without any synchronization.

Therefor PUMA is based on an efficient, kernel-level remote caching mechanism. It is is block device, file system and hypervisor agnostic. Moreover, it can operate both locally and remotely. We show through extensive experimentations that our mechanism allows applications to use memory on remote VMs to boost their performance. We also show that a VM can retrieve its own memory efficiently when needed. PUMA can help to bring a step further the notion of elasticity in clouds.

References

[Altschul 1990] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[Barham 2003] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5.

[Birke 2013] Robert Birke, Andrej Podzimek, Lydia Y Chen, and Evgenia Smirni. State-of-the-practice in data center virtualization: Toward a better understanding of vm usage. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[Bolze 2006] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, and Noredine Melab. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[Capitulino 2013] Luiz Capitulino. Automatic memory ballooning, 2013. KVM Forum.

[Dahlin 1994] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[Har'El 2013] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and scalable paravirtual I/O system. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 231–242, 2013.

[Hemminger 2005] Stephen Hemminger. Netem-emulating real networks in the lab. In *Proceedings of the 2005 Linux Conference Australia, Canberra, Australia*, 2005.

[Hines 2009] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *5th International Conference on Virtual Execution Environments, VEE 2009*, pages 51–60, 2009.

[Hwang 2014] Jinho Hwang, Ahsen Uppal, Timothy Wood, and Howie Huang. Mortar: Filling the gaps in data center memory. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, page 53–64, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2764-0.

[IBM 2010] IBM. Best practices for kvm. White Paper, Nov. 2010.

[Jaleel 2010] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 151–162, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7.

[Katcher 1997] Jeffrey Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, 1997.

[Kim 2011] Hwanju Kim, Heeseung Jo, and Joonwon Lee. Xhive: Efficient cooperative caching for virtual machines. *Computers, IEEE Transactions on*, 60(1):106–119, 2011.

[Kivity 2007] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[Magenheimer 2009] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel. Transcendent memory and linux. In *Proceedings of the 11th Linux Symposium*, pages 191–200, Montréal, Québec, Canada, 2009.

[Magenheimer 2012] Dan Magenheimer. Zcache and ramster. In *Linux Storage, Filesystem and Memory Management Summit*, 2012.

[Miller 2012] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. KSM++: using I/O-based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12)*, 2012.

[Miłós 2009] Grzegorz Miłós, Derek G Murray, Steven Hand, and Michael A Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 USENIX Annual technical conference*, pages 1–14. USENIX Association, 2009.

[Russell 2008] Rusty Russell. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42 (5):95–103, July 2008. ISSN 0163-5980.

[Salomie 2013] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 337–350, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2.

[Schopp 2006] Joel H. Schopp, Keir Fraser, and Martine J. Silbermann. Resizing memory with balloons and hotplug. In *Proceedings of the Linux Symposium*, volume 2, pages 313–319, 2006.

[Velte 2010] Anthony Velte and Toby Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010. ISBN 0071614036, 9780071614030.

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

[Zahran 2007] Mohamed Zahran, Kursad Albayraktaroglu, and Manoj Franklin. Non-inclusion property in multi-level caches revisited. *International Journal of Computers and Their Applications*, 14(2):99, 2007.