



**HAL**  
open science

# Cacol: a zero overhead and non-intrusive double caching mitigation system

Grégoire Todeschi, Boris Teabe, Alain Tchana, Daniel Hagimont

## ► To cite this version:

Grégoire Todeschi, Boris Teabe, Alain Tchana, Daniel Hagimont. Cacol: a zero overhead and non-intrusive double caching mitigation system. *Future Generation Computer Systems*, 2020, 106, pp.14-21. 10.1016/j.future.2019.11.035 . hal-02894310

**HAL Id: hal-02894310**

**<https://hal.science/hal-02894310>**

Submitted on 8 Jul 2020

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



## Open Archive Toulouse Archive Ouverte




OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible

This is an author's version published in: <https://oatao.univ-toulouse.fr/26403>

### Official URL:

<https://doi.org/10.1016/j.future.2019.11.035>

### To cite this version:

Todeschi, Grégoire  and Djongwe Teabe, Boris  and Tchana, Alain and Hagimont, Daniel  *Cacol: a zero overhead and non-intrusive double caching mitigation system.* (2020) *Future Generation Computer Systems*, 106. 14-21. ISSN 0167-739X.

Any correspondence concerning this service should be sent to the repository administrator: [tech-oatao@listes-diff.inp-toulouse.fr](mailto:tech-oatao@listes-diff.inp-toulouse.fr)

# Cacol: A zero overhead and non-intrusive double caching mitigation system

Grégoire Todeschi<sup>a,\*</sup>, Boris Teabe<sup>a</sup>, Alain Tchana<sup>b</sup>, Daniel Hagimont<sup>a</sup>

<sup>a</sup>IRIT, University of Toulouse, France

<sup>b</sup>I3S, University of Nice Sophia Antipolis, France

## A B S T R A C T

In a virtualized server, a *page cache* (which caches I/O data) is managed in both the hypervisor and the guest Operating System (OS). This leads to a well-known issue called *double caching*. Double caching is the presence of the same data in both the hypervisor page cache and guest OS page caches. Some experiments we performed show that almost 64% of the hypervisor page cache content is also present in guest page caches.

Therefore, double caching is a huge source of memory waste in virtualized servers, particularly when I/O disk-intensive workloads are executed (which is common in today's data centers). Knowing that memory is the limiting resource for workload consolidation, double caching is a critical issue.

This paper presents a novel solution (called Cacol) to avoid double caching. Unlike existing solutions, Cacol is not intrusive as it does not require guest OS modification and induces very little performance overhead for user applications. We implemented Cacol in KVM hypervisor, a very popular virtualization system. We evaluated Cacol and compared it with Linux default solutions. The results confirm the advantages (no guest modification and no overhead) of Cacol against existing solutions.

### Keywords:

Page cache

Virtualization

Double caching

## 1. Introduction

Today cloud's popularity has been made possible thanks to *virtualization* which allows the sharing of resources, thus reduce costs for Cloud clients. Cloud clients benefit from an isolated environment, called virtual machine (VM) in which runs an operating system (OS) called the guest OS. An hypervisor, a software layer similar to an OS, is responsible for the management and sharing of hardware resources between all VMs on a physical machine. The main challenge for cloud providers is to efficiently allocate resources on demand while meeting client requirements. Research has shown that memory is the most used resource in data centers (DCs) [1] and therefore a bottleneck for cloud performance. Hence, cloud providers must avoid as much as possible any kind of memory waste.

Most of modern OSs implement a cache system called *page cache* to minimize read and write latencies on storage devices by storing data into the main memory. The *page cache* is used to store in the main memory, blocks of data from storage devices to avoid expensive I/O operations (requests). Generally, in OSs and particularly in Linux, all unused memory is used for the page

cache. In virtualized environments, things get complicated in the management of the page cache. In fact, in such environments, both guest OSs and the hypervisor possess a *page cache*. The guest OS (which is a standard OS, therefore manages a page cache) in a VM usually performs I/O operations through a virtual storage device emulated by the hypervisor, the latter redirects these operations to physical storage devices. As a result, VMs' I/O requests are not only stored in their page cache but also into the hypervisor *page cache*. The outcome is a two-level *page cache* environments: (1) at the guest OS level and (2) at the hypervisor level.

Challenges tackled in this paper lies in the management of these two levels of *page cache*. We observe that there is no coordination in the handling of these two levels of cache: they act independently. As is, this inexorably leads to a well-known problem which is **double caching**. Double caching is a situation where the data in the cache at one level is also present at another level. It is also called page duplication. As we can see in Fig. 1, pages 2 and 6 of VM2 are saved in the VM *page cache*, but also in the hypervisor's one. We are therefore witnessing a double use of physical memory to store the same data. They are two direct consequences of double caching.

- **Waste of memory.** As we said earlier, memory is a critical resource in DCs. With duplicated data in two different caches, there is a waste of memory.

\* Corresponding author.

E-mail addresses: [gregoire.todeschi@enseeiht.fr](mailto:gregoire.todeschi@enseeiht.fr) (G. Todeschi), [boris.teabedjmgwe@enseeiht.fr](mailto:boris.teabedjmgwe@enseeiht.fr) (B. Teabe), [Alain.Tchana@enseeiht.fr](mailto:Alain.Tchana@enseeiht.fr) (A. Tchana), [hagimont@enseeiht.fr](mailto:hagimont@enseeiht.fr) (D. Hagimont).

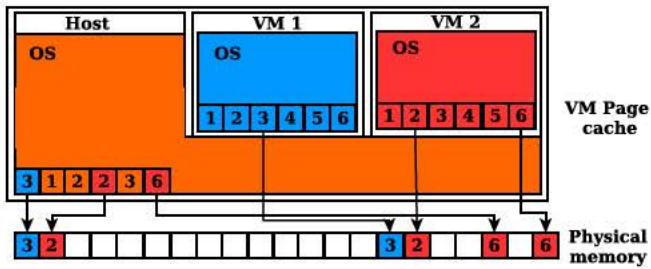


Fig. 1. Double caching in a virtualized environment.

- **Unfairness in hypervisor page cache sharing.** With no partitioning policy for the hypervisor page cache, intense use of the page cache by one or many VMs can negatively impact the performance of other VMs.

In this paper, we address the page duplication issue in the *page cache*, *Cacol*. *Cacol* is a low-overhead, dynamic, and non-intrusive cache system. The main novelties brought by *Cacol* are the following:

1. A dynamic allocation of the hypervisor page cache to each VM.
2. A non-intrusive hypervisor page cache eviction policy to overcome the double caching issue.
3. Finally, the fair sharing of the hypervisor page cache among VMs.

The rest of this article is organized as follows. In Section 2, we introduce some background elements and Section 3 presents our work assessment. Section 4 details the implementation of *Cacol*. We present the evaluation results in Section 5. Finally, we conclude the article in Section 7.

## 2. Background

### 2.1. Virtualization with KVM

KVM kernel module runs VMs as processes in the host system, and multiplexes hardware among VMs by relying on existing Linux resource-sharing mechanisms such as the scheduler, the memory management-system, the resource-accounting framework, etc. This allows KVM module to be quite small and efficient.

VM are not explicitly created and managed by the KVM module, but instead by a userspace hypervisor helper. Usually, QEMU [2] is the userspace hypervisor used with KVM. QEMU implements tasks such as VM creation, management and control. Besides, QEMU can also handle guest I/O and provide several emulated hardware devices for the VMs (such as disks, network-cards, BIOS, etc.). QEMU communicates with the KVM module using a well-defined API and ioctl interfaces. An important point to note is that VMs created by QEMU are ordinary user-space processes from the host point of view. Similarly to memory allocation for processes, QEMU allocates memory for himself and assigns it to each guest VM. Thus, for the host kernel, there is no explicit VM, but instead, userspace QEMU processes which can therefore be scheduled, swapped out, or even killed.

### 2.2. Memory management in Linux

Linux page cache [3] is a system used to speed-up I/O as it stores frequently accessed disk-blocks in memory, split into chunks called pages. The page is the smallest unit for memory management in Linux [3]. There are two different types of page

in the page cache: *File* pages referring to pages backed on disk and *Anonymous* (or *Anon*) pages which are memory allocations from various functions such as `mmap` with the `MAP_ANONYMOUS` flag, or `malloc`. Historically in Linux, those two kinds of pages were separately managed in the buffer-cache and the page cache, but they are now managed in the same place, namely the page cache. Those pages are kept indefinitely in the page cache as long as there is enough free memory. By default, the page cache uses almost all of the unused physical memory.

**Page eviction.** On memory pressure, Linux has to evict some pages from the cache in order to make room for newly allocated pages. The default eviction policy is a variant of Least Recently Used (LRU), specifically a variant of LRU-2 [4]. Thus, Linux maintains two LRU lists, respectively for active and inactive pages, so a page has to be downgraded two times to be evicted from the cache. Also, Linux manages anonymous and file pages in two separated LRU-2 lists which results in 4 lists. Despite that, anon and file pages are handled in a unified manner. Every four lists are defined per memory Cgroup. A Cgroup [5] in Linux is a feature to limit and monitor a user-defined set of processes.

The daemon in charge of page cache eviction on memory pressure is called `kswpd`. To determine how many pages to evict from each list, it relies on indicators such as cache hit or list size. Hence, when a page is evicted from the inactive list, it is swapped out if it is an anonymous one and it is emptied if it is a file page. When pages are evicted from the active list, they are downgraded to the inactive list giving them a second chance to be used.

**Writing policies.** In the operating system, several I/O paths can be used according to the policy used. Three policies exist: write-back, write-through and write-around.

*Writeback.* This policy uses normal I/O buffers that are written back to disk later by the operating system. Thus the data is first written in the page cache, and the syscall returns immediately. The data will be flushed to the physical disk later and asynchronously.

*Writethrough.* This policy guarantees that the data is committed to disk on return to userspace. In practice, it means it both writes data to the page cache and to the disk before returning. In Linux, it is done by adding the `O_SYNC` I/O flag to the write syscall.

*Write-around.* Here data are directly and synchronously written to disk, but are not kept in the page cache.

By default, Linux uses the write-back policy because it satisfies the majority of workload as it offers better performance (latency). However, it does not guarantee that all data are effectively stored to disk.

**Direct I/O.** Linux provides a way to bypass page cache by opening a file with the `O_DIRECT` flag. Then an I/O is directly sent to the disk without using the page cache. Qemu provides a way to use this flag on VM virtual disk images so a VM will not use at all any host page cache, by calling the option `cache=none` at Qemu boot.

In a KVM virtualized system, two independent page cache systems coexist, namely the host OS page cache (we note *hPC*) and the guest OS page cache (we note *gPC*). This situation leads to the *double caching* issue, assessed in the next section.

## 3. Motivations

Let us consider a VM's application which initiates a read I/O request to get *d* from the disk. The following steps are performed, summarized in Fig. 2, focusing on the case when *d* is requested for the first time.

- (1) The request is taken into account by the guest OS which first looks if its page cache contains *d*. If *d* is not present in the guest page cache, the disk driver within the guest is solicited.

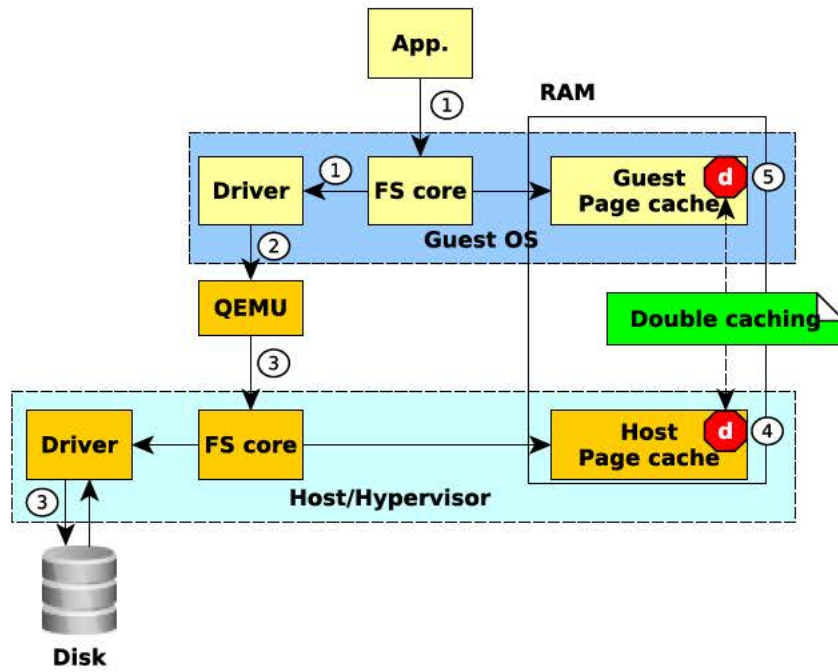


Fig. 2. Illustration of the double caching issue:  $d$  is cached in both guest page cache and hypervisor page cache.

**Table 1**  
Benchmark list used for performing experiments.

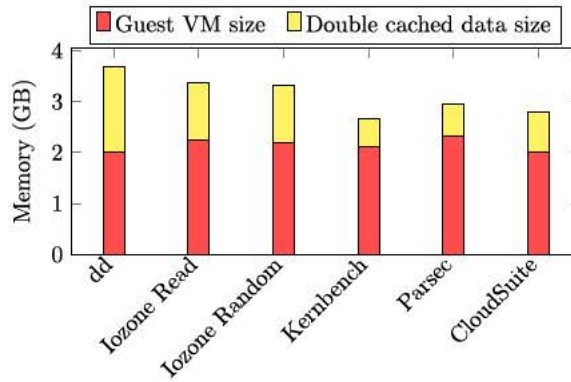
Workload	Description
Sequential read	iozone is used to measure sequential reads [6]. The file size is superior to guest cache size.
Random operations	iozone is used to measure a mixed flow of random operations [6].
CloudSuite	We use the Data caching bench, which consists of a Memcached data caching server, simulating the behavior of a Twitter caching server using a twitter dataset. The metric of interest is the throughput expressed as the number of requests served per second [7,8].
Parsec	Parsec is a benchmark suite composed of multithreaded programs [9]. We use raytrace, a real-time raytracing with native input.
dd	Use unix command dd to perform a sequential write of a big random generated file (larger than guest cache size) to /dev/null.
gzip	Use unix command gzip to perform a compression of Linux source code.
kernbench	A Linux kernel (4.0) is compiled with the kernbench script [10]. It compiles with allnoconfig. The metric of interest is the compiling time.
Filebench	Filebench [11] is a file system and storage benchmark, and run a pre-defined webserver workload. It is configured to serve 200 000 entries.

- (2) The guest disk driver tries to read the virtual disk, which traps within the QEMU process (inside the hypervisor) associated with the actual VM. In fact, the guest VM's disk is emulated by a host userspace QEMU process, which performs I/O requests on the hard disk on behalf of the guest.
- (3) As any Linux process, the I/O request issued by QEMU is taken into account by the host OS which first checks its page cache. If  $d$  is not present in the hypervisor page cache, the hard disk driver is solicited to perform the I/O request.
- (4) On data reception, the host OS stores  $d$  within its page cache and also sends it to QEMU, which in turn makes  $d$  available for the guest OS (via KVM).

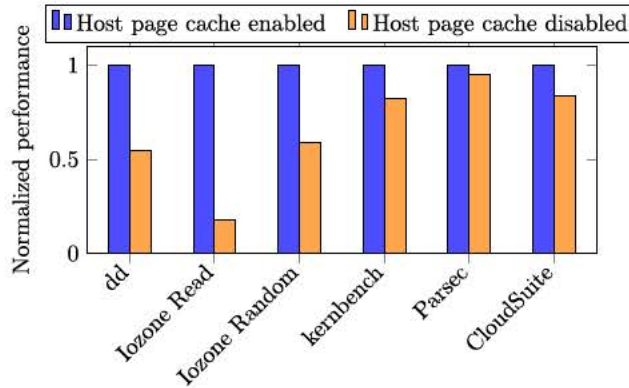
- (5) For the guest, this is a conventional disk read, and hence upon reception of  $d$ , the guest OS stores it within its page cache before sending a copy to the application.

One can see that  $d$  is cached twice (inside the host page cache and the guest page cache): this is the double caching issue [12]. The latter is a source of memory waste in virtualized DCs, knowing that memory is the limiting resource in the cloud when doing consolidation [13].

To assess the importance of this issue, we performed a set of experiments for measuring the amount of wasted memory due to double caching. We used a VM having 2 GB of memory and 4 virtual CPUs. For each experiment, the application runs alone



(a) Added memory usage (yellow) on host due to double cached data for different benchmarks



(b) Impact of host page cache on benchmarks performance (higher is better).

Fig. 3. Double caching issue illustration: added memory usage and benchmarks performance. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

on the server. Application types are detailed in Table 1. More details about the experimental environment are given in Section 5. Fig. 3(a) presents the evaluation results of double cached data. We can see that the amount of wasted memory varies from 840 MB (for random operations) up to 1680 MB (for dd), depending on the I/O request intensity. This amount is significant when compared to the VM memory size (2 GB). Considering a large scale cluster which runs an extensive number of VMs hosting similar applications, the amount of wasted memory would be massive. A projection to large scale DCs such as Alibaba, Google or Amazon clouds which run millions of VMs shows that the waste is tremendous.

A simple solution to address this issue consists in disabling the host page cache. However, this solution negatively impacts applications' performance, as presented in Fig. 3(b). We can observe a performance degradation from 5% (for Parsec raytrace workload) to up to 79% (for Iozone reading), showing that this solution is not applicable. This paper presents a solution, called Cacol, to the issue of double caching. Cacol avoids double caching while preserving applications' performance.

#### 4. Contributions

To address the double caching issue, we propose Cacol, a page cache eviction policy in the hypervisor. In this section, we first describe Cacol in Section 4.1, then in Section 4.2, we present inside on the implementation in Linux.

##### 4.1. Overall description

In the hypervisor page cache, we distinguish two types of pages: frequently accessed pages by the guest and less frequently accessed pages. Frequently accessed pages are not only stored in hypervisor page cache but also in the guest page cache due to the LFU management. These pages are likely to be duplicated in the two level of caches. The idea behind Cacol is to identify these frequently used pages and to not store them in the hypervisor cache. To do so, Cacol rely on the read and write activity on pages. When a VM requests a page for the first time, Cacol does not keep it in the hypervisor page cache. If the VM asks for this page a second time, Cacol stores the pages in the hypervisor cache. This is because, this means that the guest cache will not keep it for long and will probably request it soon, thus making it a good candidate for caching in the guest.

Cacol tracks page accesses by the VM, and as long as the VM asks for the same page again, it keeps it in the host cache. The host cache is now populated with the most requested and recent pages. Cacol therefore relies on page access frequency.

If a page is no longer accessed after a period, defined algorithmically, then it is removed from the host cache. This means that either the VM has kept it in its own cache, or the VM no longer needs it, and in both cases it is not necessary to keep this page in the host cache.

Also, Cacol needs to guarantee fairness between VMs, as one VM doing a lot of I/O will have a significant part of the host cache allocated to it, if not all.

## 4.2. Implementation

One of the difficulty encounters during the implementation was to simply modify Linux kernel as the default eviction policy is tightly tied to the whole memory management system. Apart from that, other difficulties discussed here are how to identify and differentiate correctly a page from a VM to other pages, how to keep track the number of time a page is accessed, and how to define and ensure fairness between VMs.

**Identification.** Cacol needs to identify pages that belong to a VM without accessing the VM. Therefore, Cacol stores a reference to the virtual disk image `inode` in a list when a VM starts. Thus, when a syscall occurs, it can check if the calling `inode` is in the list, and therefore identify the VM.

**Tracking access.** Cacol needs to count how many time a page have been accessed so it can apply its policy. To do that, Cacol uses a hash table to store references to VM pages. This hash table is defined in the `address_space` structure, which is a linear representation of a file or a block device. A file is divided into pages of size of 4096 bytes, so each page is defined by a unique offset, or more simply put, by an index ranging from 0 to  $file\_size\_in\_bytes/4096 - 1$ . Hence, to keep track of VM pages access with the hash table, Cacol uses the page index as the key, and stores the number of accesses as the value.

On a page request, Cacol checks the value associated with the page index in the table. There are two possibilities from there:

- It is 0, thus the page is requested for the first time. Its count is updated to 1, and the page is served to whoever requested it but it is not kept in the cache.
- It is 1 or more, meaning the page has already been accessed in the past. The page is stored in the cache if not already present, and it is served to the requester.

It is important to note that the whole `address_space` structure is deleted when the file or block device is closed, thus resetting the access count. In our case, this is not an issue because a Qemu process always keeps its virtual disk open while it is running.

**LFU management.** As Cacol is implemented on the host side, it is used as a second chance cache. Thus, we decide to use a Least Frequently Used (LFU) algorithm to order pages in the cache as it yields better performance [14].

One issue with frequency-based eviction policy is when a page is accessed a lot of time during a short period, and then is never accessed again. Such page will stay inside the cache longer than it should. To alleviate this problem, during scan to evict pages from cache, if a scanned page cannot be evicted due to its elevated count, Cacol decreases its access count and puts it back in the cache. Thus, after a finite numbers of scan, it will be finally evicted.

**Fairness.** To guarantee host page cache fairness between each VM, Cacol needs to preserve a minimum part of host page cache each VM can use and not letting one VM use all of the cache. Thus, LFU management of the cache is divided and managed on per VM basis.

Memory is also controlled to not be entirely used by one VM. For instance, the remaining free memory is split into two equal parts, the first one would provides usable page cache for all process, and the second one will be shared fairly among all the VM. For example, 4 VM running on the same host would have 12.5% of the page cache available exclusively to them. Those exclusivities materialize as low watermarks that Cacol cannot reclaim above.

On memory pressure, Cacol reclaims pages from cache. To do so, Cacol first has to choose from which VM to reclaim. It excludes VM which have reached their low watermark, as explained previously. Then, Cacol selects the least recently used VM to evict page from, until there is enough free memory or that particular

VM reaches its watermark. A VM is considered the least recently used if its most recently accessed page have been seen before other VMs most recently used page.

**Runtime management.** In addition to this, we added an interface in Linux to be able to change the cache eviction policy at runtime. The default policy set and implemented in Linux is LRU-2 which is a derivative of LRU-K [4].

## 5. Evaluations

This section presents Cacol evaluation. Objectives of this evaluation are twofold:

1. to show that Cacol can reduce the number of duplicates pages between the two-level page caches.
2. to show that the fairness policy implemented by Cacol allows the fair sharing of the hypervisor page cache across all VMs.

### 5.1. Experimental environment and methodology

**Experimental environment.** Our setup is composed of one machine with 8 GB of memory and 8 Intel i7-4800MQ cores. The guest OS is an Ubuntu server 16.04 with a Linux kernel 4.4.0. The host runs an Ubuntu 16.04, Qemu version 2.11.50, and a modified Linux kernel 4.4.0 that incorporates Cacol.

Table 1 presents the list of used benchmarks. We compared Cacol with two solutions:

- the default implementation of the page cache in KVM (noted *Default*),
- and a naive solution (noted *Naive*) which consists in disabling the host page cache.

Otherwise specified, the experimental protocol is as follows. Each benchmark is executed alone, meaning that it is the only application inside the VM and the latter runs alone on the physical machine. The VM is configured with 4 vCPUs and 4 GB of memory unless otherwise specified. Each experiment is repeated 10 times.

**Evaluation metrics.** The evaluation metrics are as follows: (1) the amount of memory wasted due to double caching, (2) and application performances in VMs.

**Profiling host page cache.** Cacol includes a small kernel module to profile and get statistics from the hypervisor page cache. The most important point is to acquire statistics exclusively from VM activities. To do so, Cacol counts page access directly in I/O kernel code path and exposes them to the kernel module. This might create an overhead to do such an accounting, but we used the accounting tool provided by Linux kernel, therefore the induce overhead is nil.

We also used the `vmtouch` [15] utility to double-check VM activity and presence in cache. It gives the number of pages the VM virtual disk has in cache via the `mincore` syscall.

### 5.2. Results with single benchmark

**Memory waste due to double caching and application performances.** Fig. 4 presents the amount of wasted memory due to double caching and Fig. 5 presents the performance of our benchmarks. Higher is better for Fig. 5 interpretation. Error bars represent the standard deviation of multiple executions of the same benchmark for both figures. Obviously, the naive solution leads to zero memory waste and there are no duplicate pages because it does not use the hypervisor page cache, this is why we did not present it on Fig. 4. On Fig. 4, we can observe that Cacol leads to almost zero memory waste for some workloads; `dd` and `gzip`. For the other benchmarks despite Cacol, there are still duplicated pages in caches. To ensure non-intrusiveness in VMs, Cacol only relies on access information available in the hypervisor on guest page caches (Section 4.1). With these informations

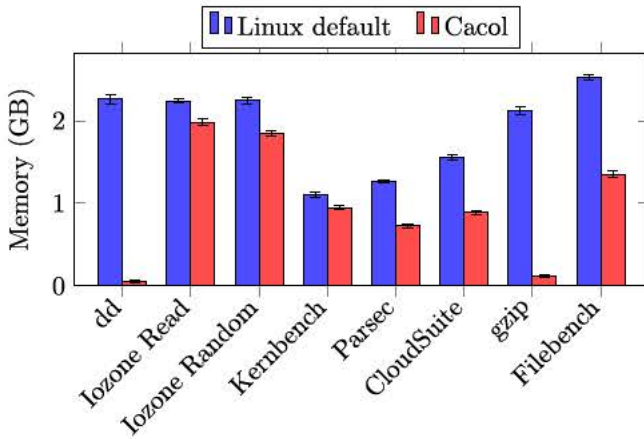


Fig. 4. Duplicate page cache size.

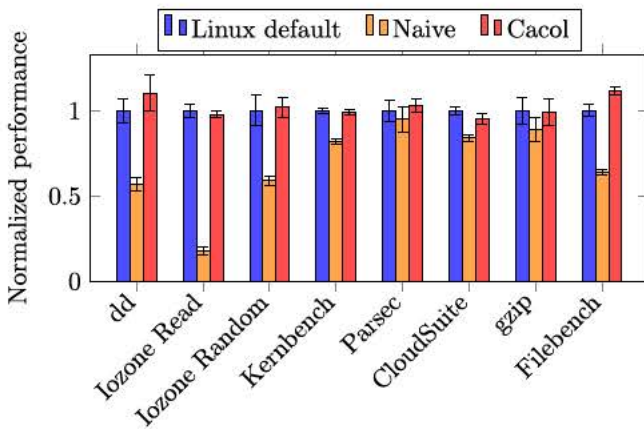


Fig. 5. Normalized performance against Linux default policy.

Cacol cannot precisely identify all pages present into guest OS caches. This explained why for some benchmarks there are still duplicated pages. Despite that, we can note on Fig. 4 that the default Linux eviction policy can waste up to 76% of cache when compared to Cacol.

The impact on the application performance depends on the benchmark working set size and the guest OS page cache size. To illustrate, let us consider on one hand a benchmark with a working set higher than the guest OS page cache. With Cacol, data stored in the hypervisor page cache will be different from those stored in the guest OS cache, therefore this increases the total size of the page cache used by the VM. For this type of benchmark, the performance will be improved because the overall hit ratio into the cache (Guest + hypervisor) will increase. This is the case with dd and Filebench benchmarks. We observe up to 11% speed-up for these benchmarks (see Fig. 5). On the other hand, if the benchmark working set is lower than the guest page cache, although Cacol will avoid data duplication, there will be little impact on benchmark performances. This is what we observe with other benchmarks. Regarding the naive solution (not using page cache in the hypervisor) this leads to a disastrous performance drop. As we can see on Fig. 5, the naive solution performs 45% worse than the Linux default policy or 50% than Cacol for the dd workload.

**Cacol overhead.** Memory overhead is the additional memory Cacol uses to run. For each page, Cacol uses an int of 4 bytes to store page access frequency. For 4096B page size, this represents an increase of 0.1% of the total memory. Moreover, there is a hash table associated with each VM to keep track of pages accessed

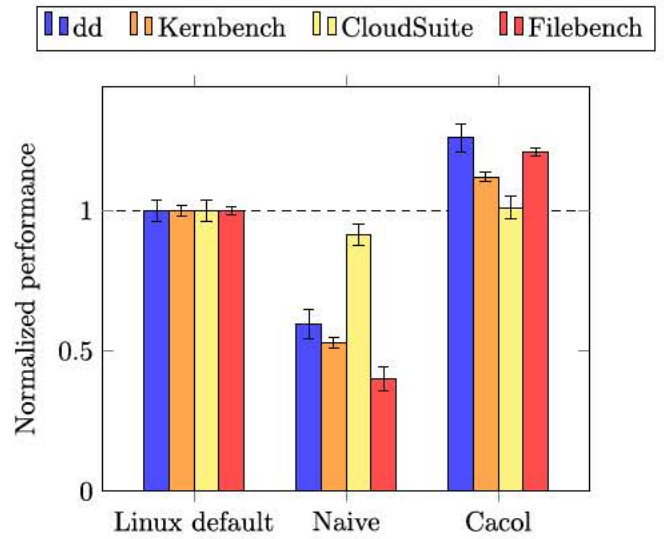


Fig. 6. Performance of 4 VMs running different benchmarks concurrently.

in the past but that are not currently present in the cache. Each potential VM page is indexed by its offset (Section 4.2), an 8 byte integer, and points to an 8 byte pointer. Overall, it represents a 0.4% memory increase of all hard disk memory. On an 50 GB virtual disk, Cacol can potentially add up to 200 MB of memory. This maximum is reached if the whole virtual disk is scanned and loaded entirely during the VM lifetime. For now, we do not limit this data structure in size, but it could eventually become a memory bottleneck and it is an improvement track for future works.

### 5.3. Results with colocated benchmarks

**Colocated VMs performance.** In this evaluation, we run simultaneously 4 VMs with 4 different applications which are dd, Kernbench, CloudSuite, and Filebench. Each VM is provisioned with 1.5 GB of memory and 1 vCPU. Results are gathered in Fig. 6. Memory is here the scarcer resource and thus the bottleneck of those memory intensive applications. We can see that Cacol yields better performance on each benchmark than default Linux. For instance, Filebench performance is improved by 21% with Cacol which is better than when it is run alone as shown in Fig. 5.

Performance improvement can be explained by the better use of memory resource. Duplicate pages are removed from the cache, and this benefits directly to other applications. In this context of large memory pressure, a duplicate page will supersede useful page from other VM because of the least recently algorithm used by Linux.

**Fairness between VMs.** Intensive I/O workloads can significantly impact cache utilization for other VMs, increasing miss ratio and thus leading to way more I/O to disk than necessary.

To highlight fairness between VMs, we run two of them simultaneously, both configured with 2 GB of memory and 2 vCPUs. One VM performs intensive I/O operations (dd) while the other performs kernel compiling (with -j2 option, using both of its vCPUs). Fig. 7 shows how host cache is consumed when those benchmarks are running. Table 2 shows the results.

As shown in Fig. 7(a), benchmark dd is doing more intensive I/O operations than kernbench. In a default configuration, dd would take almost all host page cache, up to 94% of total host page cache available. This impacts other colocated VMs performance.



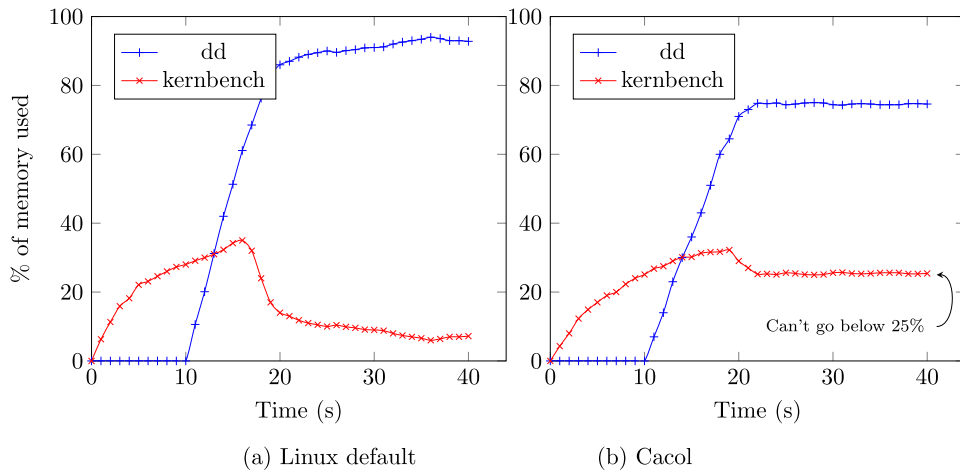


Fig. 7. Host cache usage in percentage with 2 VMs running concurrently.

Table 2

Two VMs running concurrently and impact on each other.

	Workload dd (Read speed)	Kernbench (Compile time)
Linux	46.9 MB/s	1536 s (std dev 2.27)
Cacol	46.2 MB/s	1427 s (std dev 2.85)

Cacol supervises cache utilization as described in Section 4.2, and prevents flooding of host page cache by one VM. Fig. 7(b) highlights this, as we can see the low watermark that guarantees each VM a proportion of host cache. For instance, we can see around 22 s that kernbench hits this watermark, and then pages from dd start being reclaimed.

In terms of performance, Table 2 show that kernbench performance is improved by 7.6%, because it has more memory resources available. However, dd performance is degraded by only 1.5%. Hence, Cacol sacrificed a little bit of an intensive workload performance to improve fairness.

Overall, resources are better shared among all VMs which guarantees better fairness between them in the use of host page cache.

## 6. Related work

A solution is considered intrusive if it needs to modify the VM operating system (guest OS). This is the case for example for exclusive cache [12] which provides a demote operation for VM to call the host. Intrusiveness can also be defined differently and is stated in this case.

Singleton [16] is a KVM-based system addressing the problem of double caching. Singleton relies on KSM [17] (Kernel Samepage Merging) to identify the duplicated pages. It scans the memory at regular intervals, keeps a hash of each page in a table and when it detects a duplicate, the system merges the two pages to keep only one copy and tags this page as Copy-On-Write. This solution is intrusive because it scans the memory of the VM.

Per-VM page cache [18] proposes to partition the file system cache (the host cache) into several ones, one for each VM on the physical machine. This reduces interference in the host page cache. [18] also allows you to define a policy for each cache. However, it does not provide a module to infer the type of policy to use for each VM, which must be defined by the administrator.

Sky [19] is an extension of a VM monitoring system. [19] gathers clues about file size, meta-data, and clues about the content of files, by intercepting system calls. This allows Sky to give priority on which page to put in the cache first.

Moirai [20] is a system that takes into account VM load profiles and provides tools for the cloud provider to manage the datacenter-wide cache infrastructure. It also allows you to define a cache replacement policy for each layer, i.e. in the virtual machine, on the physical host, and on the shared file system (NFS). Like Per-VM [18], Moirai relies on the administrator to configure each layers of the cache.

Multi-Cache [21] is a multi-layer cache management system that uses a combination of different types of cache to ensure a minimum quality of service. In particular, it suggests an optimal combination of cache types for different speed disks (e.g. HDD or SSD) to ensure a client defined quality of service. Multi-cache focuses primarily on optimizing latency from the raw performance of hardware components.

The works presented here try in some way to answer the problem of double caching. However, the solutions chosen are either intrusive for the guest OS, or do not have dynamic cache management following the VM activity.

## 7. Conclusion

This paper presented Cacol, a non-intrusive and dynamic cache management policy to mitigate the double caching issue in virtualized environments. Cacol uses a frequency-based eviction policy and isolation between identified virtual machine pages in the cache to achieve a significant reduction of duplicate pages. Using various workloads and benchmarks, we demonstrated that our solution yields similar performance for virtual machines' applications while shrinking cache utilization significantly. Fairness is also preserved between virtual machines which do not interfere with each others.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

- [1] N. Vlad, T. Boris, F. Leon, T. Alain, H. Daniel, StopGap: elastic VMs to enhance server consolidation, in: Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3–7, 2017, 2017, pp. 358–363, <http://dx.doi.org/10.1145/3019612.3019626>, URL <http://doi.acm.org/10.1145/3019612.3019626>.
- [2] F. Bellard, QEMU, a fast and portable dynamic translator, in: USENIX Annual Technical Conference, FREENIX Track, Vol. 41, 2005, p. 46.
- [3] R. Love, *Linux Kernel Development (Novell Press)*, Novell Press, 2005.

- [4] E.J. O'Neil, P.E. O'Neil, G. Weikum, The LRU-K page replacement algorithm for database disk buffering, SIGMOD Rec. (1993) <http://dx.doi.org/10.1145/170036.170081>, URL <http://doi.acm.org/10.1145/170036.170081>.
- [5] Linux programmer's manual CGROUPS(7), 2018, URL <http://man7.org/linux/man-pages/man7/cgroups.7.html>.
- [6] IOzone, IOzone benchmark, 2016, URL <http://www.iozone.org/>.
- [7] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A.D. Popescu, A. Ailamaki, B. Falsafi, Clearing the clouds: A study of emerging scale-out workloads on modern hardware, in: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2012, URL <http://infoscience.epfl.ch/record/173764>.
- [8] T. Palit, Y. Shen, M. Ferdman, Demystifying cloud benchmarking, in: 2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2016, pp. 122–132.
- [9] C. Bienia, *Benchmarking Modern Multiprocessors* (Ph.D. thesis), Princeton University, 2011.
- [10] C. Kolivas, Kernbench v0.50, 2012, URL <http://ck.kolivas.org/apps/kernbench/kernbench-0.50/>.
- [11] V. Tarasov, Filebench, 2019, URL <https://github.com/filebench/filebench>.
- [12] T.M. Wong, J. Wilkes, My cache or yours? Making storage more exclusive, in: Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference, ATEC '02, USENIX Association, 2002, URL <http://dl.acm.org/citation.cfm?id=647057.713858>.
- [13] J. Park, Q. Wang, J. Li, C. Lai, T. Zhu, C. Pu, Performance interference of memory thrashing in virtualized cloud environments: A study of consolidated n-tier applications, in: 2016 IEEE 9th International Conference on Cloud Computing, CLOUD, 2016, pp. 276–283, <http://dx.doi.org/10.1109/CLOUD.2016.0045>.
- [14] Y. Zhou, J. Philbin, K. Li, The multi-queue replacement algorithm for second level buffer caches, in: USENIX Annual Technical Conference, General Track, 2001, pp. 91–104.
- [15] D. Hoyte, Vmtouch - the virtual memory toucher, 2018, URL <https://hoYTECH.com/vmtouch/>.
- [16] P. Sharma, P. Kulkarni, Singleton: System-wide page deduplication in virtual environments, in: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC '12, ACM, 2012, <http://dx.doi.org/10.1145/2287076.2287081>, URL <http://doi.acm.org/10.1145/2287076.2287081>.
- [17] A. Arcangeli, I. Eidus, C. Wright, Increasing memory density by using KSM, in: *Proceedings of the Linux Symposium, Citeseer*, 2009, pp. 19–28.
- [18] P. Sharma, P. Kulkarni, P. Shenoy, Per-VM page cache partitioning for cloud computing platforms, in: 2016 8th International Conference on Communication Systems and Networks, COMSNETS, 2016, <http://dx.doi.org/10.1109/COMSNETS.2016.7439971>.
- [19] L. Arulraj, A.C. Arpacı-Dusseau, R.H. Arpacı-Dusseau, Improving virtualized storage performance with sky, SIGPLAN Not. (2017) <http://dx.doi.org/10.1145/3140607.3050755>, URL <http://doi.acm.org/10.1145/3140607.3050755>.
- [20] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, T. Talpey, Software-defined caching: Managing caches in multi-tenant data centers, in: ACM Symposium on Cloud Computing, SOCC 2015, ACM, 2015, URL <https://www.microsoft.com/en-us/research/publication/software-defined-caching-managing-caches-in-multi-tenant-data-centers/>.
- [21] S. Rajasekaran, S. Duan, W. Zhang, T. Wood, Multi-cache: Dynamic, efficient partitioning for multi-tier caches in consolidated VM environments, in: 2016 IEEE International Conference on Cloud Engineering, IC2E, 2016, <http://dx.doi.org/10.1109/IC2E.2016.10>.



**Grégoire Todeschi** received a Master degree from Polytechnic National Institute of Toulouse in 2016. He then started a Ph.D. course in 2016 at Polytechnic National Institute of Toulouse, France. His main research interests are in Virtualization, Cloud Computing, and Operating System.



**Boris Teabe** received his Ph.D. from Polytechnic National Institute of Toulouse in 2017. Since 2017 he is a research engineer at IRIT lab, Toulouse France. His main research interests are in Virtualization, Cloud Computing, and Operating System.



**Alain Tchana** received his Ph.D. in computer science in 2011, at the IRIT laboratory, Polytechnic National Institute of Toulouse, France. In September 2013, he joined Polytechnic National Institute of Toulouse as a Associate Professor. Since 2018, he is a Professor at University of Nice Sophia Antipolis. His main research interests are in Virtualization, Cloud Computing, and Operating System.



**Daniel Hagimont** is a Professor at Polytechnic National Institute of Toulouse, France and a member of the IRIT laboratory, where he leads a group working on operating systems, distributed systems and middleware. He received a Ph.D. from Polytechnic National Institute of Grenoble, France in 1993. After a postdoctorate at the University of British Columbia, Vancouver, Canada in 1994, he joined INRIA Grenoble in 1995. He took his position of Professor in Toulouse in 2005.